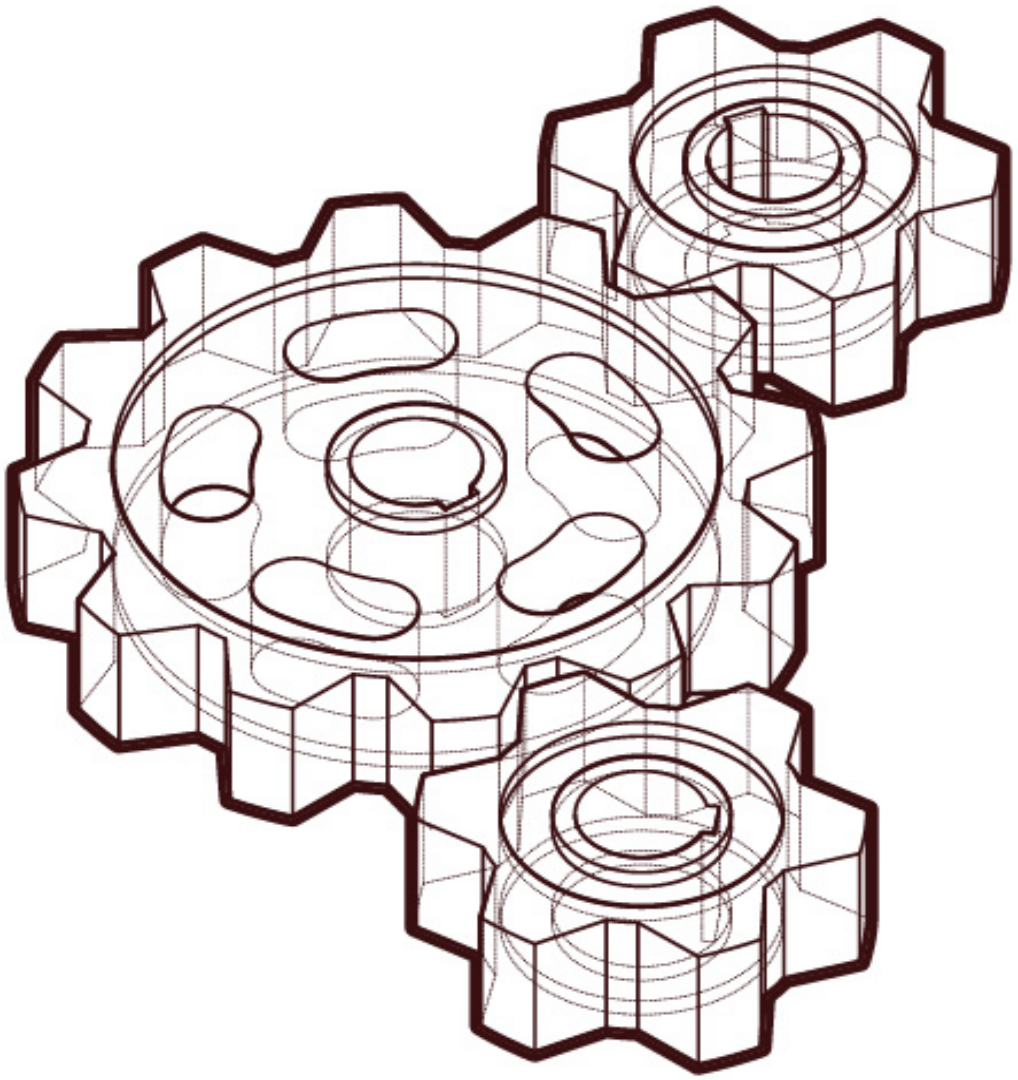
Robot framework

Automação versátil e consistente para testes



Casa do Código | alura

THOMÁS DANIEL VIEIRA

Sumário

- ISBN
- Sobre este livro
- Sobre o autor
- 1. Introdução
- 2. Primeiros passos com Robot Framework
- 3. Escalando o projeto: arquitetura de Page Objects
- 4. Lidando com a massa de dados dos testes
- 5. Testes de API
- 6. Estruturas de controle
- 7. Testes para dispositivos móveis
- 8. Bibliotecas do usuário
- 9. Boas práticas e guia de estilos
- 10. Para saber mais
- 11. Referências
- 12. Anexo I: Técnicas de mapeamento de elementos HTML
- 13. Anexo II: Camadas de abstração para projetos de automação

ISBN

Impresso: 978-85-5519-345-3

Digital: 978-85-5519-344-6

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Vieira, Thomás Daniel
Robot framework : automação versátil e consistente para testes / Thomás Daniel Vieira. -- São Paulo : AOVS Sistemas de Informática, 2023.

ISBN 978-85-5519-345-3

1. Automação 2. Framework (Programa de computador) 3. Robôs - Programação I. Título.

23-164734 CDD-005.1

Índices para catálogo sistemático:

1. Programação de computadores 005.1

Eliane de Freitas Leite - Bibliotecária - CRB 8/8415

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre este livro

Há algum tempo, testar software de forma completamente manual deixou de ser eficiente. A **automação de testes** é uma estratégia necessária, visto que times ágeis precisam de *feedback* rápido e constante sobre a qualidade de novas versões de seus sistemas. A obtenção da entrega contínua seria inviável se fosse preciso gastar dias ou semanas realizando os testes para cada nova versão.

Segundo Robert Martin, os testes de aceitação sempre devem ser automatizados, por razões de custo e eficiência. Caso contrário, não será viável que o time faça pequenas entregas frequentes de novas versões.

De forma simplificada, automatizar testes significa desenvolver um "robô" que testará automaticamente as funcionalidades de um sistema, sem necessidade de intervenção humana, com objetivo de garantir a qualidade desse software de forma mais eficiente.

Neste livro você aprenderá a construir passo a passo seus primeiros scripts de testes automatizados utilizando o **Robot Framework**, uma das ferramentas sugeridas por Markus Gärtner em seu livro *ATDD by Example* para o Desenvolvimento Guiado por Testes de Aceitação.

O Robot Framework é um framework genérico para construção de scripts automatizados baseado em abordagem *keyword-driven* utilizando sintaxe tabular, que o torna legível e amigável. O framework pode ser utilizado para automação de testes e processos. É uma ferramenta de código aberto, extensível e escrita com linguagem Python. Existem bibliotecas para construção de testes em diferentes níveis e tecnologias, bem como para prover ferramentas auxiliares como geração randômica de massa de dados, conexão com banco de dados e manipulação de arquivos. Tudo isso torna o Robot uma opção completa e capaz de solucionar bem praticamente qualquer desafio em automação.

Esta é uma ferramenta estável e de fácil aprendizagem. Não obstante, novas versões com melhorias e novas funcionalidades seguem sendo desenvolvidas. Utilizaremos neste livro a versão 6.0, a última versão estável em janeiro de 2023. No capítulo de introdução deste livro, mostraremos como preparar e configurar todo o ambiente de desenvolvimento necessário.

A documentação existente é extensa e bem detalhada, não só da ferramenta em si como também de suas bibliotecas, e pode ser encontrada em: robotframework.org/robotframework.

A Robot Framework Foundation, consórcio que suporta a tecnologia, possui 47 empresas, e este número aumentou 10 vezes entre 2015 e 2020, de forma que o investimento na manutenção da ferramenta também cresceu consideravelmente, mantendo-se estável mesmo durante a pandemia de Covid-19.

Este livro destina-se a quem deseja começar a construir seus próprios testes automatizados e já possui conhecimento prévio de lógica de programação e qualidade de software. O objetivo aqui não é esmiuçar todas as funcionalidades da ferramenta, mas aprender o necessário para construir uma solução profissional e consistente.

Sempre haverá indicação de onde encontrar informações mais detalhadas sobre os assuntos abordados ao longo do livro. Por serem padrão de mercado, muitos dos conceitos que serão apresentados sobre projetos de automação se aplicam a qualquer ferramenta com essa finalidade — e relembraremos isso no decorrer da nossa jornada de aprendizado.

Para facilitar a prática e experimentação, os exemplos utilizados neste livro podem ser encontrados nas *branches* do repositório https://github.com/thvieira/robot-framework-demo, onde serão atualizados quando necessário. Pratique o máximo que puder: fazer somente a leitura não é suficiente para o aprendizado efetivo de

ferramentas que envolvem programação, e muitas dúvidas só aparecem quando colocamos as mãos na massa.

Bons estudos!

Sobre o autor

Meu nome é **Thomás Daniel Vieira**, nasci em Charqueadas, no Rio Grande do Sul, onde me formei técnico em Informática pelo Instituto Federal Sul-rio-grandense (IFSul), após meteórica carreira como aprendiz eletromecânico. Sou bacharel em Ciências da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) e possuo especialização em Engenharia de Software pela Unisinos e especialização em Produção Editorial pela LabPub. Atualmente, estudo Design Editorial no Núcleo de Estratégias e Políticas Editoriais (NESPE).

Desde 2010, já trabalhei como desenvolvedor web, desenvolvedor mobile, desenvolvedor back-end, engenheiro de testes automatizados, dentre outras coisas. Atualmente, trabalho como especialista em testes automatizados para empresas de Portugal e ministro treinamentos na área de programação e de automação em Porto Alegre e arredores.

Devido ao gosto pelos livros, minha formação como engenheiro de software é muito influenciada por clássicos de Robert Martin, Martin Fowler, Kent Beck, Markus Gärtner, GoF e outros livros da própria Casa do Código. Eles serão citados muitas vezes como referência no meu texto.

Como a carreira de tecnologia é dinâmica, trabalhei com diversas ferramentas e linguagens de programação, sem me especializar em nenhuma delas. Optei por estudar os padrões de arquitetura de software e de qualidade de código para estar preparado para qualquer desafio.

Além disso, paralelamente às minhas atividades como engenheiro de software, trabalho como editor na grande pequena Editora Coragem, focada em questões sociais e literatura latino-americana. Por isso, este livro foi escrito, em boa parte, entre os pampas e a Cordilheira dos Andes.

Sinta-se à vontade para me contatar:

LinkedIn: linkedin.com/in/thomasvieira.

Medium: medium.com/@thomas.vieira.

E-mail: thvieira.rs@gmail.com.

Capítulo 1
Introdução

O Robot Framework é uma biblioteca de código aberto desenvolvida em Python para *Robotic Process Automation* (RPA). O projeto é mantido pelo consórcio Robot Framework Foundation (https://robotframework.org/foundation), um grupo que conta com a participação de grandes empresas globais e que cresce consideravelmente, aumentando, consequentemente, o investimento na manutenção e na evolução da ferramenta. O grupo realiza anualmente uma conferência internacional chamada Robocon (https://robocon.io/), na qual se apresentam as novidades das últimas versões e se discutem novas funcionalidades, boas práticas, projetos inovadores e formas de difundir o aprendizado na ferramenta. As palestras são gravadas e estão disponíveis no site da conferência.

A ferramenta foi recomendada por Markus Gärtner em seu livro *ATDD by Example* (2012) e, assim como suas concorrentes, possui vantagens e desvantagens, a depender do contexto. Uma das principais vantagens é a versatilidade, dada a possibilidade de construção de scripts para diversos tipos e níveis de testes, como testes de interface web, mobile, desktop, testes de API de serviços, testes em banco de dados, dentre muitos outros. Isso é possível simplesmente instalando as bibliotecas complementares do Robot conforme a necessidade. O framework possui boa comunidade ativa e um grande ecossistema em torno dele, resultando em várias bibliotecas e ferramentas genéricas que são desenvolvidas como projetos separados.

Além disso, o usuário pode escrever os cenários de teste em sintaxe tabular, com linguagem humana amigável e de fácil aprendizagem, podendo ou não ser estruturada com Gherkin. Isso é fundamental para a legibilidade do código-fonte, aspecto essencial em projetos de

automação. Outros pontos fundamentais em projetos de automação são o reaproveitamento e a manutenibilidade do código, e por isso é possível criar diretivas personalizadas reaproveitáveis. Sua estrutura modular facilita a concepção de uma arquitetura consistente para o projeto. Raramente não encontraremos uma biblioteca já existente para atender nossa necessidade, mas, se for preciso, podemos criar novas bibliotecas personalizadas através da linguagem Python.

Ainda pensando nos princípios fundamentais para um bom projeto de automação, há suporte para linha de comando, o que traz boa aderência com as ferramentas de integração contínua. Ao final das execuções, o Robot fornece automaticamente relatórios e logs em formato HTML completos e fáceis de compreender, a partir dos quais é possível obter diferentes métricas e dados relativos aos testes que foram executados.

Abordaremos neste livro sobre como construir scripts automatizados utilizando Robot Framework, trabalhando de maneira prática e passo a passo a construção de projetos em uma situação hipotética, semelhante a situações reais e frequentes do mercado na área de Engenharia de Software.

Para ter acesso à documentação oficial, histórico de versões, links para comunidade, informações sobre o projeto, lista de bibliotecas disponíveis e até mesmo à loja virtual de lembrancinhas, visite o site oficial: https://robotframework.org.

O problema

Consideraremos que estamos entrando em um time de desenvolvimento ágil que está construindo uma loja virtual para uma livraria. Nosso desafio é garantir a qualidade do produto entregue.

Durante as primeiras iterações, as versões entregues ainda possuíam poucas funcionalidades, portanto a equipe realizava alguns testes manualmente mesmo. Entretanto, depois de alguns meses, o

sistema já apresentava várias novas funcionalidades e o trabalho tornou-se custoso e demorado demais.

Todo time ágil de desenvolvimento de software deseja praticar entregas frequentes de pequenas versões do seu sistema, seja para uma simples correção de bug ou para novas funcionalidades. Como garantir a qualidade das novas versões em tempo hábil?

A solução

Uma maneira de resolver esse problema é investir em **automação de testes**. De forma simplificada, isso significa desenvolver um "robô" que testará automaticamente as funcionalidades de um sistema, sem necessidade de intervenção humana, com objetivo de garantir a qualidade desse software de forma mais eficiente.

Os testes devem fornecer um parecer objetivo quanto à qualidade do software que está sendo entregue, garantindo que as funcionalidades cumpram os requisitos funcionais e não funcionais que se esperam delas. Testes automatizados não devem depender de intervenção humana para que sejam corretamente executados. A suíte de testes deve ser executada através de ferramentas de integração contínua, seja através de agendamentos sistemáticos a cada período de tempo ou por gatilhos configurados no repositório de versionamento do código fonte de produção. O resultado deve ser sempre assertivo, ou seja, deve emitir um parecer PASSOU ou FALHOU, sem necessidade de interpretação humana para obter essa resposta. A entrega contínua depende desse princípio.

1.1 Tipos e níveis de teste

Seguindo a recomendação de Robert C. Martin em seu livro *The Clean Coder* (2011), trabalharemos para automatizar ao menos todos os testes de aceitação de cada nova funcionalidade, criando

uma bateria de testes de regressão automatizada. Os **testes de regressão** garantem que a implementação das novas versões não introduzirão defeitos nas funcionalidades que já existiam nas versões anteriores.

Os testes de aceitação validam os requisitos de negócio que devem ser contemplados para que uma funcionalidade possa ser considerada entregue pelo time. Somente quando todos os critérios de aceitação estiverem contemplados é que os testes passarão e a funcionalidade estará pronta para ser implantada, atendendo às expectativas do cliente. Segundo Martin, os testes de aceitação sempre devem ser automatizados e rodar em um fluxo de integração contínua, por razões de custo e eficiência. Caso contrário, não será viável que o time faça pequenas entregas frequentes de novas versões. Mas, afinal, devemos automatizar todo e qualquer caso de teste do sistema?

1.2 Quando e o que automatizar

Automatizar tudo e qualquer cenário de teste não é a solução de todos os problemas de qualidade de um time de desenvolvimento de software. Devemos investir tempo e esforço em automatizar apenas o que vale a pena ser automatizado e nos gerará eficiência com o passar do tempo. Nem todo cenário de teste é assim. É necessário que a pessoa engenheira de testes automatizados faça a análise das funcionalidades do sistema com o intuito de identificar quais cenários devem ser automatizados, e a cobertura de testes automatizados ideal varia em cada empresa, sistema e contexto.

A **cobertura de testes automatizados** diz respeito à porcentagem de requisitos e funcionalidades do sistema que pode ser testada automaticamente, mas é preciso sensatez ao lidar com essa métrica. Segundo Martin Fowler em seu artigo *Test coverage* (2012a), definir determinado nível de cobertura de testes como meta

pode provocar a construção de testes de baixa qualidade apenas para atingir essa porcentagem.

Tradicionalmente, os testes da bateria de regressão possuem centenas (às vezes milhares!) de cenários de teste mapeados. Em outro contexto, em um time de desenvolvimento com pequenas entregas contínuas, o tempo disponível talvez seja pequeno para implementação de scripts automatizados para tudo o que está sendo entregue. E talvez essas funcionalidades recém-entregues sofram alterações nas próximas *sprints*, o que é normal. Nesses casos, não existe uma bala de prata, mas é sensato começar pensando nos testes dos serviços de *back-end*.

Testes redundantes ou repetitivos são grandes candidatos para a automação. Testes que precisam de observação humana, como determinar um bom visual estético de um website ou verificar se a navegação do menu é intuitiva, devem continuar manuais. Testes manuais sempre serão importantes — mesmo com a automação, existem cenários nos quais a execução deles é necessária.

Como citado anteriormente, os testes de aceitação devem ser automatizados, caso contrário, não será viável que o time faça pequenas entregas frequentes de novas versões, ou seja, a homologação automatizada é fundamental para praticar *entrega contínua*. Esse é o raciocínio essencial para eleger o que devemos priorizar para automação.

Reúna a equipe, identifique os requisitos de aceitação de cada funcionalidade e comece garantindo que estes cenários serão automatizados, sendo executados a cada entrega do time. Acredite: quem atinge esse nível de maturidade no trabalho com testes automatizados possui uma solução de qualidade mais consistente do que times que possuem centenas de cenários automatizados, mas que nem sempre são confiáveis.

Os benefícios da automação são particulares para cada equipe, isso não pode ser generalizado. Saber identificar quais cenários de teste

trarão retorno ao investimento feito na criação dos scripts automatizados é fundamental, afinal, não somos robôs de criar robôs.

O sistema da livraria virtual possui funcionalidades vitais, tais como carrinho de compras, cadastro e pagamento. Outra funcionalidades como o formulário de contato, lista de desejos e avaliação dos produtos não são essenciais para o negócio. Dentre as funcionalidades mais importantes, podemos iniciar o trabalho pelas implementações mais simples, de maneira que estaremos entregando algo de valor em períodos o mais breves possíveis. Considerando a estabilidade da funcionalidade — afinal, quanto menos manutenção ela sofrer, menos manutenção nosso script de teste também sofrerá — convém iniciarmos pela implementação dos testes da funcionalidade de cadastro.

Como estamos em um contexto no qual já existem muitas funcionalidades entregues, é preciso organizar um *backlog* de automação. E a técnica empregada aqui foi a da **Matriz de Valor x Complexidade**, na qual escrevo um *post-it* para cada funcionalidade e as organizo em um quadro marcado por dois eixos: *complexidade de implementação do teste* e *importância para o negócio e para o usuário*. As primeiras funcionalidades a serem puxadas para a *sprint* serão as que estiverem no quadrante superior, que indica alto valor para o negócio e menor complexidade de implementação.

Martin Fowler demonstra em seu blog e Robert Martin detalha em seus livros o conceito da **pirâmide de testes** automatizados apresentado por Mike Cohen. Essa é uma maneira gráfica de demonstrar a distribuição ideal do esforço em construção de diferentes níveis de teste, considerando custo e complexidade de cada um.

Um conjunto de milhares de testes unitários pode ser executado em menos de um minuto e, como são construídos pelos próprios programadores durante o ciclo de desenvolvimento, o custo de implementação é baixo. Sendo assim, os testes unitários são a base da pirâmide, portanto, estes testes devem cobrir o mais próximo possível de 100% do sistema. Na camada intermediária estão os testes de componentes, geralmente em nível de API ou outras formas de integração, a depender da arquitetura do sistema em questão. Geralmente, centenas de casos de testes de API são executados em poucos minutos e sua implementação é simples, como veremos em alguns capítulos mais adiante. Os testes de interface devem garantir os critérios de aceitação das funcionalidades, cobrindo algo em torno de 20% das funcionalidades do sistema ou até menos, pois esses testes são mais complexos de implementar e manter, além de tomarem mais tempo para serem executados, pois simulam as ações do usuário de abrir o navegador ou aplicativo, clicar em botões, esperar a tela carregar etc.

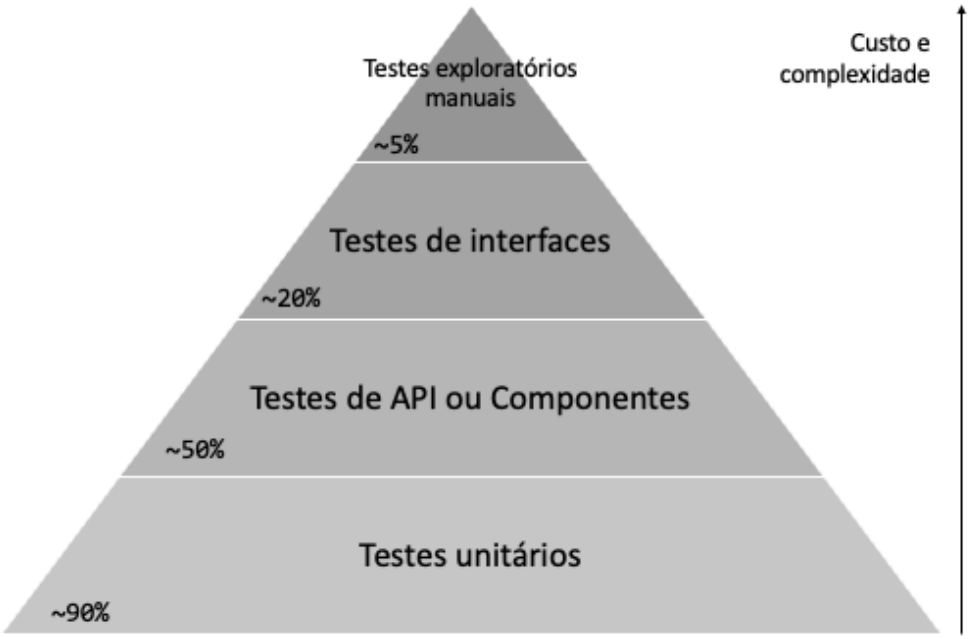


Figura 1.1: A pirâmide de testes.

Em uma situação real, talvez a implementação dos testes automatizados iniciaria em nível de API, entretanto, por questões

didáticas do livro, iniciaremos implementando testes de interface.

Após analisar, eleger e priorizar os cenários de teste que serão automatizados nas próximas iterações, iniciaremos a construção do nosso projeto de automação de testes utilizando Robot Framework. O primeiro passo é preparar nosso ambiente de desenvolvimento, para então começarmos com os primeiros scripts.

1.3 Preparando o ambiente de desenvolvimento

Para trabalhar com Robot Framework, tudo que precisamos ter instalado é o Python, um navegador web qualquer e nosso editor de código-fonte favorito, como Atom, VSCode, PyCharm etc.

Para os exemplos deste livro, recomendo utilizar o Python na versão 3 ou superior (disponível em www.python.org/downloads). Para verificar se o Python está corretamente instalado na sua máquina e em qual versão, abra o terminal de comandos e execute:

```
$ python --version
```

Instalação da biblioteca do Robot

Como dito anteriormente, o Robot Framework é uma biblioteca desenvolvida em Python, portanto, podemos fazer a instalação dela através do `pip`, o gerenciador de pacotes do Python, com o comando a seguir:

```
$ pip install robotframework
```

Aqui utilizaremos a versão 6.0, a última versão estável. Para verificar qual versão foi instalada, utilize o comando:

```
$ robot --version
```

Um tutorial mais detalhado sobre esse procedimento pode ser encontrado em https://robotframework.org/?tab=1#getting-started.

Instalando a biblioteca do Selenium

A interação com o navegador é feita através do Selenium (www.selenium.dev), a ferramenta de código aberto mais popular para automação de testes web. Precisamos instalar a biblioteca do Selenium para Robot, também através do `pip`.

```
$ pip install --upgrade robotframework-seleniumlibrary
```

A documentação completa da biblioteca está em https://robotframework.org/SeleniumLibrary. É importante termos esse recurso sempre à mão, pois as possibilidades são inúmeras e não precisamos decorar todos os comandos existentes. Além do mais, a documentação é bastante completa e de fácil compreensão.

Configurando o WebDriver

O Selenium requer um executável chamado *WebDriver* para a manipulação do navegador. Esse executável deve ser baixado de acordo com o navegador no qual se deseja executar os testes, observando a compatibilidade de versões entre eles. Existem *WebDrivers* para os principais navegadores do mercado, como Mozilla Firefox, Google Chrome, Opera Browser e Safari.

Utilizaremos o Mozilla Firefox em nossas demonstrações, por preferência pessoal. Escolha o navegador de sua preferência e instale o *WebDriver* correspondente. O *WebDriver* do Selenium para Mozilla Firefox chama-se *geckodriver* e pode ser encontrado em https://github.com/mozilla/geckodriver/releases.

Após o download, extraia o arquivo executável, caso esteja compactado. Se você estiver utilizando Windows, escolha um diretório arbitrário para armazenar os *WebDrivers* e insira o caminho desse diretório nas variáveis de ambiente do sistema. Caso esteja

utilizando Linux ou MacOS, simplesmente mova o arquivo para o diretório `/usr/bin/` .

Para verificar se tudo ocorreu bem, utilize o comando a seguir:

```
$ geckodriver --version
```

Agora temos tudo pronto para desenvolver scripts com Robot Framework!

Capítulo 2
Primeiros passos com Robot Framework

Para nos familiarizarmos com a ferramenta, vamos implementar uma simples busca no Google.

A primeira coisa que precisamos fazer é criar um diretório para nosso projeto, por exemplo, `google-test/`, tendo em mente a convenção de utilizar o sufixo "test" na nomenclatura de projetos de teste. Em seguida, criaremos um arquivo `.robot` onde implementaremos nosso script, que deve ser nomeado com a funcionalidade à qual se destina a suíte de testes que vamos criar, juntamente do sufixo `_test`, por exemplo, `google_search_test.robot`. Com isso, estamos informando claramente que, neste arquivo, estarão os cenários de teste da suíte de testes da funcionalidade de busca do Google.

> **Código de testes também deve ser limpo**
>
> Como Robot é uma biblioteca do Python, seguimos o mesmo guia de estilos dessa tecnologia, que podem ser encontrados em peps.python.org/pep-0008. Falaremos mais disso adiante.

Os arquivos `.robot` são estruturados em seções, sendo que nenhuma delas é obrigatória. Ao longo da construção dos nossos scripts, apresentaremos as principais delas. Iniciaremos a construção do script importando as bibliotecas necessárias para a construção dos testes. A declaração de bibliotecas deve estar na seção *Settings*, a qual normalmente se encontra no topo do documento. A sintaxe da declaração de sessões utiliza três asteriscos antes e depois do nome da sessão.

Como visto anteriormente, a ferramenta que nos permite interagir com o navegador é o Selenium, então a primeira coisa que faremos

é importar essa biblioteca em nosso script para que possamos utilizar seus recursos.

Como código-fonte vale mais do que mil palavras, vamos lá:

```
*** Settings ***
Library  SeleniumLibrary
```

Note que existem dois espaços entre a palavra reservada `Library` e o nome da biblioteca `SeleniumLibrary` ! É assim mesmo que tem que ser.

> **Atenção aos espaços e tabulações**
>
> O Robot utiliza sintaxe posicional, baseada em espaços ou tabs. Os elementos são separados por tabs ou dois espaços ou mais.
>
> Isso existe para permitir o alinhamento dos comandos e parâmetros, com o intuito de facilitar a legibilidade do código. No entanto, pode causar bastante confusão. Em alguns momentos nosso código não funcionará como esperado e o problema pode estar sendo causado por não termos percebido espaços a mais ou a menos. O *highlight* do editor ajuda, mas é importante estar atento.

Agora podemos criar nosso primeiro cenário de teste, que será uma simples busca no Google para consultar a previsão do tempo. Os passos dos cenários de teste devem estar na seção "Test Cases". Nosso primeiro cenário se chamará "Previsão do tempo em Porto Alegre".

Naturalmente, a primeira ação do script deve ser abrir o navegador. Faremos isso com o comando `Open browser` .

```
*** Settings ***
Library  SeleniumLibrary

*** Test Cases ***
```

```
Cenário: Previsão do tempo em Porto Alegre
    Open browser
```

Com apenas essas linhas de código, podemos executar nosso script pela primeira vez e conferir se está tudo funcionando corretamente. Basta utilizar a seguinte linha de comando via terminal, estando no diretório raiz do projeto.

```
$ robot google_search_test.robot
```

Várias coisas novas podem ter acontecido neste instante. Analisaremos a situação por partes.

> Utilize o parâmetro `-d` para definir o diretório onde guardar os logs e relatórios da execução do teste, por exemplo, `reports/`. Neste caso, o comando deve ser executado da seguinte forma: `$ robot -d reports/ google_search_test.robot`.

Caso o navegador Mozilla Firefox não esteja instalado ou se o Geckodriver não tiver sido corretamente configurado, a execução falhará devido a um `WebDriverException` por não ter encontrado o *'geckodriver' executable*. Caso o Firefox e o Geckodriver estejam instalados e configurados, uma janela do navegador deve ter sido aberta na tela. Isso acontece porque não informamos para o Robot qual navegador desejamos utilizar, então foi utilizado o navegador padrão, que é o Mozilla Firefox. Se quisermos executar em outro navegador, devemos definir isso explicitamente nos parâmetros do comando `Open browser`.

```
*** Settings ***
Library  SeleniumLibrary

*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser  browser=chrome
```

Os parâmetros podem ser nomeados para facilitar a compreensão do código, caso necessário, mas isso não é obrigatório se os parâmetros estiverem na mesma ordem de argumentos esperada pelo comando.

Perceba também que, ao final da execução, alguns arquivos foram criados na raiz do nosso projeto, `log.html`, `output.html` e `report.html`. Estes são os logs e relatórios da execução!

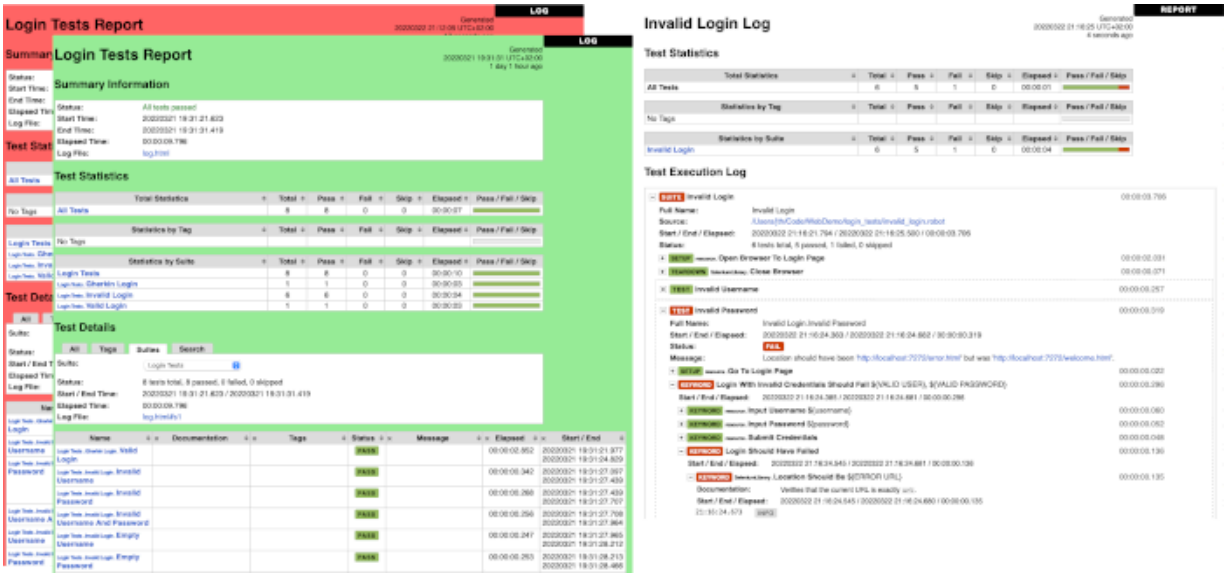


Figura 2.1: Logs e reports do Robot Framework.

Para que esses arquivos não fiquem soltos no projeto, existe a convenção de alocá-los no diretório `reports/`, o que deve ser feito automaticamente adicionando um parâmetro ao comando de execução dos scripts:

```
$ robot -d reports/ google_search_test.robot
```

Outro detalhe da execução é que o navegador foi aberto e assim permaneceu. Esse é um comportamento que não é desejado, pois o script deve fechar a janela automaticamente ao final. Para isso, existe o comando `Close browser`.

```
*** Settings ***
Library  SeleniumLibrary
```

```
*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser   browser=chrome
    Close browser
```

Seguindo em frente, vamos acessar o site do Google utilizando o comando `Go to`. Pode parecer estranho utilizar o comando `Go to`, mas no Robot esse comando serve para acessar determinada página web através de uma URL, bem diferente do clássico e desagradável desvio de fluxo incondicional.

```
*** Settings ***
Library  SeleniumLibrary

*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser   browser=firefox
    Go To          http://www.google.com.br
    Close Browser
```

Alternativamente, podemos abreviar os comandos `Open browser` e `Go to` em um só.

```
*** Settings ***
Library  SeleniumLibrary

*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser  url=http://www.google.com.br  browser=firefox
    Close Browser
```

Agora precisamos inserir um texto no campo de busca do Google e teclar `Enter`. Utilizaremos o comando `Input text` para digitar o texto da busca e o comando `Press key` para teclar `Enter`, mas primeiro precisamos mapear o elemento na tela. Utilizando a ferramenta Firebug do Firefox ou Inspector do Chrome, é possível identificar o elemento HTML que representa o campo de texto. Podemos notar que ele possui o atributo `name="q"`, que servirá para identificá-lo.

Atributos *name* ou *id* são perfeitos para o mapeamento dos elementos.

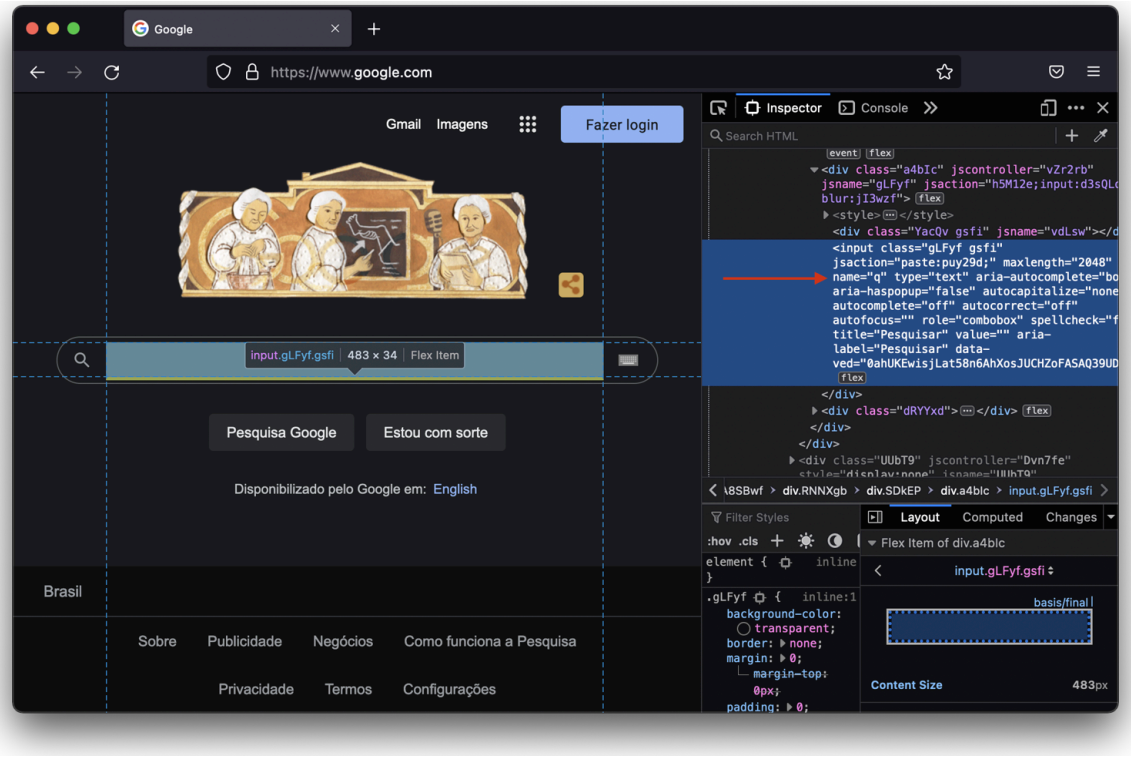


Figura 2.2: Mapeamento do elemento campo de texto.

E o código ficará assim:

```
*** Settings ***
Library  SeleniumLibrary

*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser  browser=firefox
    Go To         url=http://www.google.com.br
    Input Text    name="q"    Previsão do tempo em Porto Alegre
    Press Keys    name="q"    RETURN
    Close Browser
```

Execute o script. Ops! O código falhou. O que fizemos de errado? Analisemos a mensagem de erro exibida tanto no console quanto no relatório: `"'[name=""q""]' is not a valid selector"` . O problema está

em nosso mapeamento, e isso se deve a um erro bem simples, mas que muitas vezes passa despercebido. Não devemos utilizar aspas na construção do parâmetro localizador. Em vez de `name="q"` , o correto é suprimir as aspas e utilizar `name=q` .

Outro detalhe que podemos constatar após essa falha é que o nosso navegador não foi encerrado automaticamente após a execução como esperávamos. Isso ocorreu porque o fluxo de execução foi interrompido antes do comando `Close browser` . Felizmente, existe uma forma de garantir que o navegador seja fechado ao fim da execução mesmo que aconteça alguma falha, através do comando `Teardown` , que deve estar na seção de `Settings` e serve para definir que determinado comando seja executado incondicionalmente após cada cenário da suíte de testes.

```
*** Settings ***
Library  SeleniumLibrary

Test Teardown  Close Browser

*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser  browser=firefox
    Go To         url=http://www.google.com.br
    Input Text    name=q    Previsão do tempo em Porto Alegre
    Press Keys    name=q    RETURN
```

Neste momento, está faltando somente o nosso script realizar a validação do resultado de nossa ação. Esperamos encontrar nos resultados da busca a previsão do tempo para os próximos dias em Porto Alegre. Existem diversas formas de fazer essa validação, entretanto, concluiremos nosso exemplo com uma validação simples através da mensagem exibida na tela.

Para validar que determinada mensagem foi exibida na tela após a execução de uma ação, utilizaremos o comando `Wait until page contains` . Esse comando faz com que o WebDriver aguarde por alguns segundos até que o carregamento da página seja concluído e

o texto esperado seja exibido na tela. Caso o tempo esgote sem a mensagem aparecer, o teste falha. É possível definir quantos segundos são aceitáveis através do parâmetro `timeout`, que é opcional.

Para conhecer todos os comandos para validações disponíveis, acesse a documentação da biblioteca do Selenium para Robot: https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html.

O código-fonte com a validação que acabamos de inserir fica assim:

```
*** Settings ***
Library  SeleniumLibrary

Test Teardown  Close Browser

*** Test Cases ***
Cenário: Previsão do tempo em Porto Alegre
    Open browser  browser=firefox
    Go To         url=http://www.google.com.br
    Input Text    name=q      Previsão do tempo em Porto Alegre
    Press Keys    name=q      RETURN
    Wait Until Page Contains  Previsão do tempo para os próximos 15 dias
em Porto Alegre
```

Execute o script e analise seus resultados. Faça alterações, experimente alterar os dados do teste e rode novamente. Nosso robô é capaz de realizar uma busca no Google!

Assim, concluímos nosso primeiro script com Robot. Estamos prontos para seguir em frente e construir testes mais elaborados para nossa livraria virtual.

2.1 Construindo o primeiro teste no sistema da livraria

O código-fonte do site da livraria virtual que utilizaremos para os exemplos está no repositório github.com/thvieira/bookstore-demo. A seguir, a tela principal do sistema:

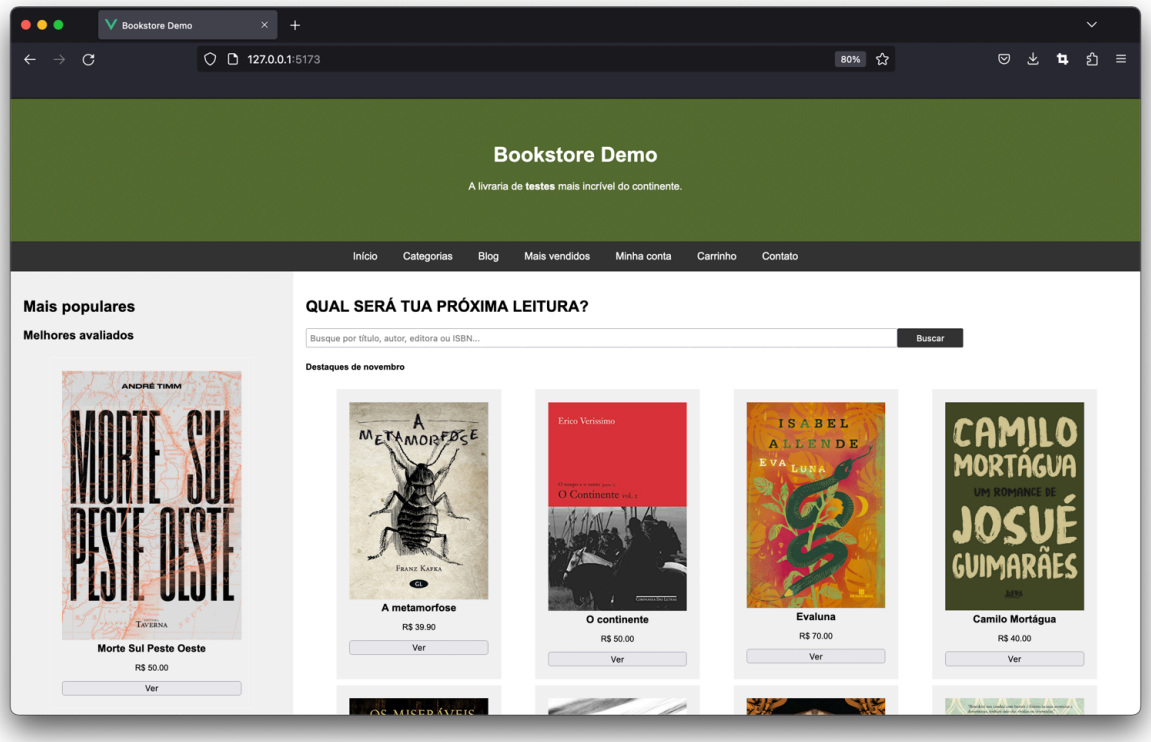


Figura 2.3: Tela principal da livraria virtual.

Agora que estamos mais familiarizados com o Robot Framework, podemos criar um diretório para nosso projeto de testes automatizados para a livraria virtual, que chamaremos de `bookstore-demo-test`.

A primeira funcionalidade que testaremos será o *Cadastro*, iniciando pelo cenário de cadastro com sucesso de pessoa física, cobrindo o requisito de que um cliente físico deve ser capaz de se cadastrar no sistema, realizar autenticação e acessar sua conta.

```
Como cliente da loja virtual, desejo criar um cadastro no sistema para
poder visualizar meus pedidos e lista de desejos.

Cenário: Cadastro com sucesso de Pessoa Física
  Dado que estou na página da loja virtual
      E acessei a área de cadastro
  Quando preencho o formulário com meus dados
      E submeto os dados do formulário
      E aceito os termos de uso
  Então devo receber uma mensagem de sucesso
      E devo visualizar a opção de acessar meus pedidos
      E devo visualizar a opção de acessar minha lista de desejos
```

Assim podemos criar o arquivo `register_test.robot`, seguindo a mesma estrutura que aprendemos a fazer para a busca no Google. Na seção `Settings` do documento, utilizamos o comando `Library` para importar a biblioteca do Selenium e o comando `Test Teardown` para fechar o navegador ao final da execução do script. Na seção `Test Cases`, criamos o nosso cenário de teste abrindo o navegador e acessando o site da loja virtual. Podemos utilizar a configuração `Documentation` na seção `Settings` para definir que este arquivo representa a suíte de testes para a funcionalidade em questão. As reticências servem para quebrarmos a linha e o comando continuar na linha seguinte. Os primeiros passos do script serão abrir o navegador e acessar o site da livraria.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Library         SeleniumLibrary

Test Teardown   Close Browser

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Open browser  browser=firefox
    Go To         url=http://tst.bookstore.com/
```

O próximo passo é acessar a área de cadastro. Utilizaremos o comando `Click link` para acessar o menu "Minha conta", que exibirá a seguinte tela:

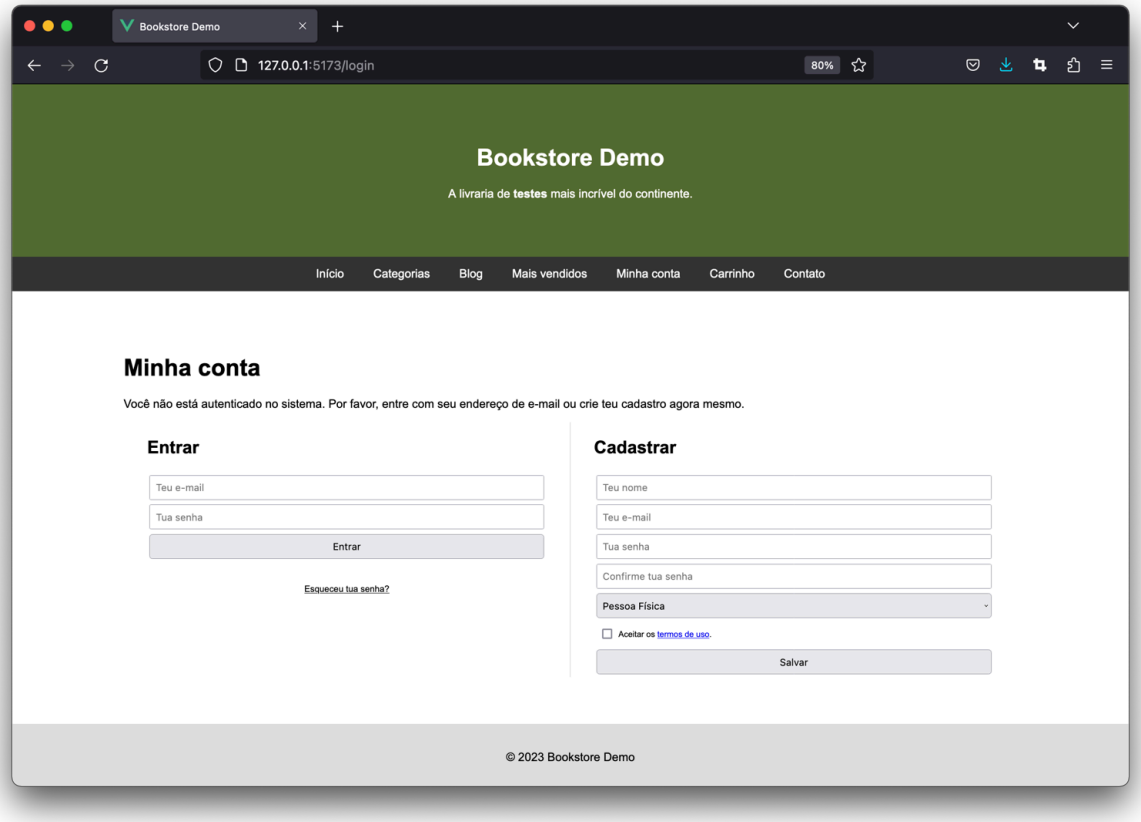


Figura 2.4: Tela de login e de cadastro.

Podemos utilizar simplesmente o texto "Minha conta" como localizador para mapear o botão, pois este comando utiliza o texto contido na tag `<a>` ou o valor do atributo `href` como estratégia padrão de localizador, diferentemente do padrão dos outros comandos que utilizam o valor dos atributos `id` e `name`.

MAPEAMENTO DOS ELEMENTOS DA PÁGINA

Existem várias estratégias de localizadores para mapear os elementos de uma página web. As mais utilizadas são `id`, `name` e `xpath`. Para definir a estratégia que deseja utilizar de forma explícita, a sintaxe deve ter a forma `estratégia: valor` ou `estratégia=valor`. A primeira opção é a mais recomendada, porque a segunda opção é idêntica à sintaxe de argumentos nomeados explicitamente, o que pode causar confusão.

Espaços entre os separadores são ignorados, de forma que `id:foo`, `id: foo` e `id : foo` são equivalentes.

A lista de estratégias de localizadores do Selenium para Robot pode ser encontrada no link robotframework.org/SeleniumLibrary ou no Anexo I, ao final deste livro.

Da mesma forma que realizamos o mapeamento do botão do menu, devemos mapear os campos de texto da página de cadastro. Utilizando o comando `Input text`, podemos preencher o formulário. Ao final, podemos clicar no botão "Salvar" ou utilizar o comando `Submit` para submeter o formulário.

O elemento ComboBox de seleção do tipo de pessoa, física ou jurídica, deve ser manipulado através de um dos três comandos a seguir, a depender da melhor abordagem:

- `Select from list by label`;
- `Select from list by value`;
- `Select from list by index`.

A abordagem "*by label*" selecionará a opção cujo conteúdo textual da tag `<option>` seja exatamente correspondente ao parâmetro da busca. A abordagem "*by value*" realizará a busca baseada no valor do atributo `value` das tags `<option>` e a abordagem "*by index*"

selecionará a opção baseada no número sequencial de acordo com a ordem das tags `<option>`, o índice, conforme demonstrado no código HTML do elemento ComboBox a seguir.

```
<select name="user-type" id="register-user-type" required>
  <option value="pf">Pessoa Física</option>    <!-- index 01 -->
  <option value="pj">Pessoa Jurídica</option>  <!-- index 02 -->
</select>
```

Utilizaremos neste exemplo a abordagem baseada no *label* das opções, para melhor legibilidade do script.

Perceba que a tela de login e cadastro, ilustrada na figura anterior, possui dois formulários, um para autenticação no sistema e outro para cadastro no sistema. Alguns campos possuem o mesmo valor no atributo `name`, e isso pode causar problemas em nossos mapeamentos. Atributos `name` podem, eventualmente, se repetir, mesmo que não seja uma situação corriqueira. Para garantir que tudo ocorrerá bem, vamos utilizar os atributos `id` como estratégia de localizador sempre que possível, pois o `id` sempre será único na página.

Note que utilizaremos um comando um pouco diferente do `Input text` para inserir os valores dos campos de senha. A única diferença é que o valor será ocultado no relatório gerado pela execução, evitando expor dados que podem ser sensíveis.

O preenchimento do *checkbox* de aceite dos termos de uso será realizado através do comando `Select Checkbox`, que funciona de forma muito parecida com um simples clique.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Library         SeleniumLibrary

Test Teardown   Close Browser
```

```
*** Test Cases ***
Scenario: Success sign in as Natural Person
    Open browser    browser=firefox
    Go To           url=http://tst.bookstore.com/
    Click Link      Minha conta
    Input Text      id: register-name           Lorem ipsum
    Input Text      id: register-mail           lorem@ipsum.com
    Input Password  id: register-pass           foo123
    Input Password  id: register-confirm-pass   foo123
    Select From List By Label  id: register-user-type  Pessoa Física
    Select Checkbox  id: terms-of-use
    Submit Form     name: register
```

Para que o script seja considerado realmente um teste, ele precisa validar algo ao final. Precisamos validar que o sistema emite uma mensagem de sucesso após o cadastro e que as opções de acessar "Meus pedidos" e "Lista de desejos" estão disponíveis para o usuário. Como podemos construir essas verificações? Analise a tela a seguir.

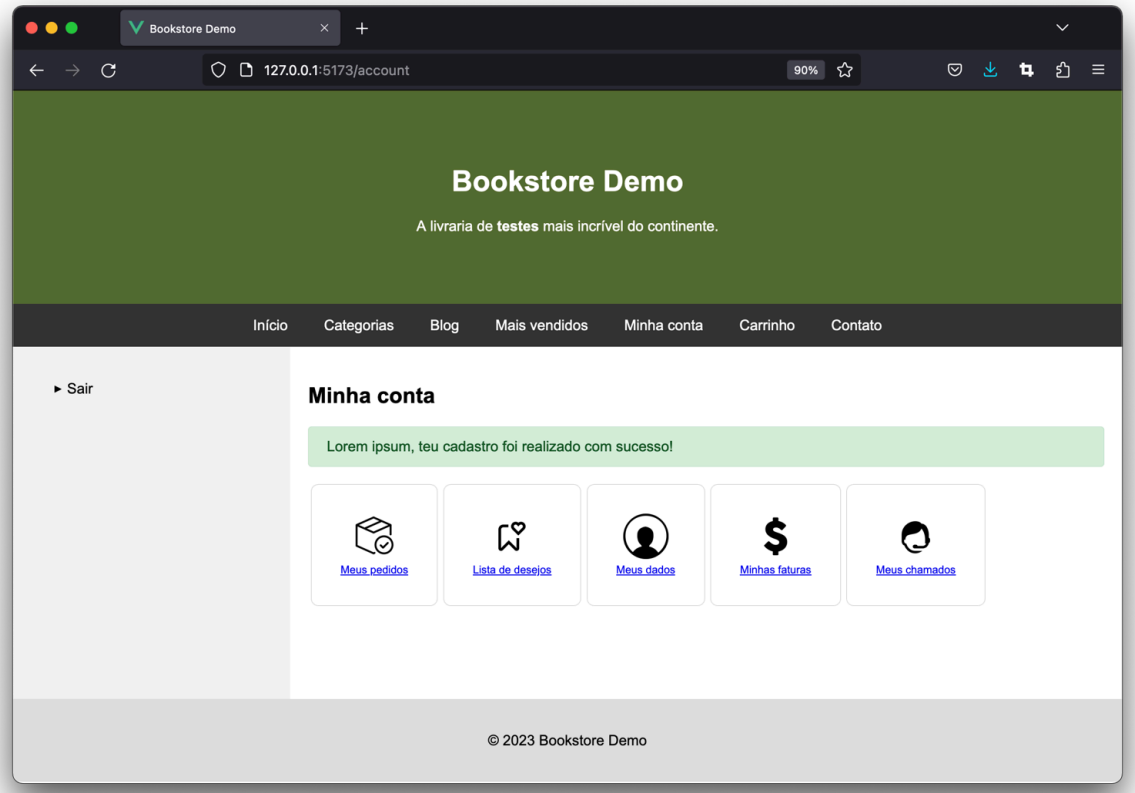


Figura 2.5: Tela de controle da conta do usuário.

As validações podem ser realizadas de diversas maneiras diferentes. Validar se a mensagem de sucesso está visível na tela pode ser feito através do comando `Wait Until Page Contains`, da mesma forma que fizemos no exemplo inicial da busca no Google. Quanto à presença das opções de acesso à lista de desejos e pedidos, podemos validar através do comando `Page Should Contain Link`, que validará se o link para determinada página está presente na tela.

As possibilidades de validação são muitas, portanto, é interessante explorar a documentação dos comandos `built-in` do Robot, em robotframework.org/latest/libraries/BuiltIn.html, e do Selenium, em robotframework.org/SeleniumLibrary.html.

```
*** Settings ***
```

```
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Library         SeleniumLibrary

Test Teardown   Close Browser

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Open browser     browser=firefox
    Go To            url=http://tst.bookstore.com/
    Click Link       Minha conta
    Input Text       id: register-name           Lorem ipsum
    Input Text       id: register-mail           lorem@ipsum.com
    Input Password   id: register-pass           foo123
    Input Password   id: register-confirm-pass   foo123
    Select From List By Label  id: register-user-type  Pessoa Física
    Select Checkbox  id: terms-of-use
    Submit Form     name: register
    Wait Until Page Contains  Lorem ipsum, teu cadastro foi realizado com
sucesso!
    Page Should Contain Link  /my-orders
    Page Should Contain Link  /my-wish-list
```

Com isso, temos nosso primeiro caso de teste funcionando, mas nosso projeto pode melhorar. Para que fique fiel ao cenário descrito em Gherkin, podemos utilizar as diretivas *Given/When/Then*, nativas do Robot Framework. Basta inserir a diretiva antes do comando que o comportamento do script não se modificará, o efeito será estritamente estético. Nesta primeira versão, a leitura do script ainda não flui de maneira tão agradável, mas veremos como melhorar esse aspecto nos próximos capítulos.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Library         SeleniumLibrary
```

```
Test Teardown   Close Browser

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given Open browser     browser=firefox
      And Go To            url=http://tst.bookstore.com/
      And Click Link       Minha conta
     When Input Text       id: register-name           Lorem ipsum
      And Input Text       id: register-mail           lorem@ipsum.com
      And Input Password   id: register-pass           foo123
      And Input Password   id: register-confirm-pass   foo123
      And Select From List By Label  id: register-user-type  Pessoa Física
      And Select Checkbox  id: terms-of-use
      And Submit Form      name: register
     Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
      And Page Should Contain Link  /my-orders
      And Page Should Contain Link  /my-wish-list
```

No próximo capítulo, nossa suíte de testes vai aumentar. Será que estamos preparados para isso?

CAPÍTULO 3
Escalando o projeto: arquitetura de Page Objects

Partindo para os próximos cenários de testes, nosso projeto começará a aumentar. No capítulo anterior, implementamos o cenário de teste para cadastro com sucesso de pessoa física. Criaremos agora o script para o cenário de teste de cadastro com sucesso de pessoa jurídica.

Como este é um caso de teste da mesma funcionalidade na qual estávamos trabalhando anteriormente, podemos escrever nosso script no mesmo arquivo, dentro da seção `Test Cases`. Além dos dados de cadastro, a única diferença entre os passos deste cenário e os passos do cenário anterior está na opção selecionada no ComboBox, em que, em vez de "Pessoa Física", devemos selecionar a opção "Pessoa Jurídica". Em vez de validar a opção de visualizar a lista de desejos, um usuário "Pessoa Jurídica" deve poder visualizar a lista de faturas. Com base no que já havíamos criado anteriormente, um código inicial ficaria assim:

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Library         SeleniumLibrary

Test Teardown   Close Browser

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given Open browser     browser=firefox
      And Go To            url=http://tst.bookstore.com/
      And Click Link       Minha conta
     When Input Text       id: register-name          Lorem ipsum
      And Input Text       id: register-mail          lorem@ipsum.com
```

```
      And Input Password   id: register-pass          foo123
      And Input Password   id: register-confirm-pass  foo123
      And Select From List By Label  id: register-user-type  Pessoa Física
      And Select Checkbox  id: terms-of-use
      And Submit Form      name: register
     Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
      And Page Should Contain Link  /my-orders
      And Page Should Contain Link  /my-wish-list

Scenario: Success sign in as Juridical Person
    Given Open browser     browser=firefox
      And Go To            url=http://tst.bookstore.com/
      And Click Link       Minha conta
     When Input Text       id: register-name          Lorem ipsum LTDA
      And Input Text       id: register-mail
contato@loremipsum.com
      And Input Password   id: register-pass          54321
      And Input Password   id: register-confirm-pass  54321
      And Select From List By Label  id: register-user-type  Pessoa
Jurídica
      And Select Checkbox  id: terms-of-use
      And Submit Form     name: register
     Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
      And Page Should Contain Link  /my-orders
      And Page Should Contain Link  /my-invoices
```

Esse script funciona, mas funcionar não é o suficiente para programadores e programadoras profissionais. Código-fonte duplicado significa arquitetura ruim e implica muitos problemas para o projeto, em especial para a manutenibilidade do teste. Sistemas desenvolvidos em uma equipe que pratica iterações e pequenas entregas receberão atualizações constantemente. Muitas vezes, quando as telas de uma funcionalidade forem alteradas, o código de testes de interface deverá receber manutenção para se adaptar à mudança. Gosto de lembrar que o código-fonte será mais vezes lido

do que escrito, conforme a afirmação de Robert C. Martin no livro *Código Limpo* (2008). É essencial que projetos de testes automatizados sejam fáceis de alterar, e isso significa que ele deve ser legível e sem duplicidade de código.

Apesar disso, escrever o código da forma mais simples possível para solucionar o problema, mesmo que isso signifique uma implementação caótica, e então melhorar a solução para que se torne bem estruturada, com uma boa arquitetura, é uma ótima abordagem de desenvolvimento, como propõe Martin Fowler no livro *Refatoração* (2020).

Vamos analisar nosso problema passo a passo para identificar de que formas podemos solucionar os problemas de baixa manutenibilidade do nosso projeto. As ações de abrir o navegador e acessar o site da livraria virtual não se repetirão somente agora, mas em todos os cenários de teste que criarmos para este sistema, então devemos utilizar um recurso nativo do Robot para que determinadas ações sejam sempre executadas **antes** de cada cenário: o `Test Setup`. Ele funciona de forma semelhante ao `Test Teardown`, que já estamos utilizando para que o robô execute determinadas ações **depois** de cada cenário. Além disso, podemos utilizar o parâmetro `url` do comando `Open browser` para que, ao abrir a janela do navegador, o robô imediatamente acesse o sistema, sem necessidade do comando `Go to`. Alteraremos ambos os cenários para aplicar essa refatoração.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Library         SeleniumLibrary

Test Setup      Open Browser  browser=firefox
url=http://tst.bookstore.com/
Test Teardown   Close Browser

*** Test Cases ***
```

```
Scenario: Success sign in as Natural Person
    Given Click Link       Minha conta
     When Input Text       id: register-name          Lorem ipsum
      And Input Text       id: register-mail          lorem@ipsum.com
      And Input Password   id: register-pass          foo123
      And Input Password   id: register-confirm-pass  foo123
      And Select From List By Label  id: register-user-type  Pessoa Física
      And Select Checkbox  id: terms-of-use
      And Submit Form      name: register
     Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
      And Page Should Contain Link  /my-orders
      And Page Should Contain Link  /my-wish-list

Scenario: Success sign in as Juridical Person
    Given Click Link       Minha conta
     When Input Text       id: register-name          Lorem ipsum LTDA
      And Input Text       id: register-mail
contato@loremipsum.com
      And Input Password   id: register-pass          54321
      And Input Password   id: register-confirm-pass  54321
      And Select From List By Label  id: register-user-type  Pessoa
Jurídica
      And Select Checkbox  id: terms-of-use
      And Submit Form      name: register
     Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
      And Page Should Contain Link  /my-orders
      And Page Should Contain Link  /my-invoices
```

Dica: utilize tags para executar apenas um caso de teste específico, ou um determinado conjunto de casos de teste. Veja como trabalhar com as tags do Robot no *Capítulo 6: Boas práticas e código limpo*.

3.1 Configurações globais do projeto

Algumas informações — como navegador, URL do sistema, dados sensíveis de autenticação, limites de timeout etc. — são definições gerais de projeto e serão utilizadas por praticamente todas as suítes de teste, portanto, devem estar isoladas em um arquivo de configuração e não dentro dos arquivos de casos de teste. Essas informações poderão mudar com a necessidade de execução em diferentes ambientes prática comum em muitas empresas que possuem, por exemplo, uma versão do sistema para *development*, *testing* e *production*.

Por enquanto, simplesmente isolaremos a definição do navegador e da URL do sistema. Criaremos para isso um arquivo chamado `config.resource`. Nele, utilizaremos a seção de `Variables` do Robot, dedicada à declaração de variáveis, para definir nosso navegador e URL do sistema.

Variáveis

No Robot Framework, as variáveis são nomeadas sempre entre chaves, por exemplo, `${var_name}`, `${GLOBAL_VAR_NAME}` e `@{fone_list}`. Note que a convenção de nomenclatura adotada para o nome de variáveis é *snake_case* e *ALL_CAPS*, no caso de constantes ou variáveis globais. As variáveis podem ser usadas como escalares, listas ou dicionários através da sintaxe `${SCALAR}`, `@{LIST}` e `&{DICT}`, respectivamente. Além disso, as variáveis de ambiente podem ser usadas diretamente com a sintaxe `%{ENV_VAR}`.

Para mais detalhes sobre a utilização de variáveis com Robot Framework, acesse o seguinte endereço: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#variables.

Nosso arquivo de configurações ficará assim:

```
*** Settings ***
Documentation  Project global definitions.

*** Variables ***
${BROWSER}     firefox
${URL}         http://tst.bookstore.com/
```

E então basta importar o arquivo no arquivo da suíte de testes através do comando `Resource` e substituir os valores fixos pelas variáveis.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.

Library         SeleniumLibrary
Resource        config.resource

Test Setup      Open Browser    ${URL}  ${BROWSER}
Test Teardown   Close Browser
```

3.2 Mapeamento de objetos

Quando as telas do sistema sob teste forem alteradas pelos desenvolvedores, existe o risco de os localizadores dos mapeamentos de alguns elementos utilizados pelos testes automatizados também mudarem, e o código-fonte do teste precisará passar por manutenção. O que acontecerá com nosso projeto neste caso?

No projeto de testes atual, teríamos que alterar os mapeamentos em mais de um local dentro do projeto. A tendência é que, com mais cenários implementados, alguns mapeamentos sejam utilizados

dezenas de vezes. Caso seja necessário realizar a mesma alteração em cada um destes locais onde o mapeamento se repete, a manutenção se tornará inviável. Devemos centralizar e reutilizar esses mapeamentos.

Substituiremos os localizadores utilizados diretamente nos comandos por variáveis, movendo essas informações para uma seção separada, `Variables`, e utilizaremos nomenclaturas que ajudem a descrever qual elemento cada localizador representa.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.

Library         SeleniumLibrary
Resource        config.resource

Test Setup      Open Browser    ${URL}  ${BROWSER}
Test Teardown   Close Browser

*** Variables ***
${MY_ACCOUNT_BUTTON}         Minha conta
${NAME_TEXT_FIELD}           id: register-name
${MAIL_TEXT_FIELD}           id: register-mail
${PASS_TEXT_FIELD}           id: register-pass
${CONFIRM_PASS_TEXT_FIELD}   id: register-confirm-pass
${BUSINESS_ENTITY_COMBOBOX}  id: register-user-type
${TERMS_OF_USE_CHECKBOX}     id: terms-of-use
${REGISTER_BUTTON}           name: register
${MY_WISH_LIST_LINK}         /my-wish-list
${MY_ORDERS_LINK}            /my-orders
${MY_INVOICES_LINK}          /my-invoices

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given Click Link                ${MY_ACCOUNT_BUTTON}
    When Input Text                 ${NAME_TEXT_FIELD}           Lorem
ipsum
```

```
    And Input Text                ${MAIL_TEXT_FIELD}
lorem@ipsum.com
    And Input Password            ${PASS_TEXT_FIELD}          foo123
    And Input Password            ${CONFIRM_PASS_TEXT_FIELD}  foo123
    And Select From List By Label ${BUSINESS_ENTITY_COMBOBOX}  Pessoa
Física
    And Select Checkbox           ${TERMS_OF_USE_CHECKBOX}
    And Submit Form               ${REGISTER_BUTTON}
   Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
    And Page Should Contain Link  ${MY_WISH_LIST_LINK}
    And Page Should Contain Link  ${MY_ORDERS_LINK}

Scenario: Success sign in as Juridical Person
   Given Click Link               ${MY_ACCOUNT_BUTTON}
   When Input Text                ${NAME_TEXT_FIELD}          Lorem
ipsum LTDA
    And Input Text                ${MAIL_TEXT_FIELD}
contact@ipsum.com
    And Input Password            ${PASS_TEXT_FIELD}          foo123
    And Input Password            ${REGISTER_PASS_TEXT_FIELD}  foo123
    And Select From List By Label ${BUSINESS_ENTITY_COMBOBOX}  Pessoa
Jurídica
    And Select Checkbox           ${TERMS_OF_USE_CHECKBOX}
    And Submit Form               ${REGISTER_BUTTON}
   Then Wait Until Page Contains  Lorem ipsum, teu cadastro foi
realizado com sucesso!
    And Page Should Contain Link  ${MY_INVOICES_LINK}
    And Page Should Contain Link  ${MY_ORDERS_LINK}
```

Com essa refatoração, de quebra melhoramos também a legibilidade do código. Ao nomear os mapeamentos dos elementos com nomenclaturas expressivas, ficou muito mais fácil identificar qual elemento aquele mapeamento representa. Eventualmente os mapeamentos serão mais complexos, de maneira que não será intuitivo identificar qual elemento da tela está sendo mapeado. Ao abstrair o mapeamento em uma variável e atribuir a ela um nome expressivo e descritivo, o trabalho de manutenção ficará mais fácil.

Entretanto, este arquivo `.robot` no qual estamos trabalhando possui mais de uma responsabilidade. Além dos cenários de teste, ele possui também o mapeamento dos objetos. Esses mapeamentos precisam ser separados em uma nova camada arquitetural em nosso projeto. Mesmo que o Robot Framework não seja orientado a objetos, nós podemos e devemos pensar na arquitetura do nosso projeto e seguir o princípio da responsabilidade única, tanto em relação aos arquivos quando a comandos que criaremos. Separaremos a lista de localizadores que acabamos de criar do script principal do caso de teste.

O padrão arquitetural que estamos montando aqui é o **Page Objects**. Seu princípio básico é encapsular o mapeamentos dos elementos da interface do sistema sob teste, seja qual for a tecnologia, e abstrair o comportamento desses elementos. A camada de abstração mais elementar é a do mapeamento dos objetos, cuja responsabilidade é estritamente encapsular os mapeamentos dos elementos da interface com os quais o robô interage.

Como o Robot Framework não é orientado a objetos, a implementação do padrão arquitetural precisará de algumas adaptações em relação às definições da documentação oficial do Selenium para atender às especificidades da tecnologia com a qual estamos trabalhando. É normal que isso aconteça, mas devemos fazer todo o possível para atenuar essas diferenças e manter nosso projeto perfeitamente compreensível para quem está acostumado a trabalhar com projetos de automação em outras linguagens.

Martin Fowler recomenda não criar um arquivo de *Page Object* para cada página ou tela do sistema, e sim para cada componente significativo do sistema, por exemplo, para o menu, para a galeria de fotos e para o rodapé. Essa abordagem evita redundâncias e facilita a localização dos mapeamentos dentro do projeto.

Devemos então criar um arquivo separado e mover nossos mapeamentos para lá. Separaremos nossos elementos em três arquivos, um para os elementos do menu, outro para a página de

cadastro e, por fim, um para os elementos da página "Minha conta". O padrão de estilos recomenda que a nomenclatura dos arquivos de *Page Object* sejam o nome da página ou componente seguido do sufixo `_page_object`. É importante notar que os arquivos auxiliares não devem possuir a extensão `.robot`, que é exclusiva para os casos de teste. Os arquivos de *Page Objects* devem utilizar a extensão `.resource`. Portanto, criaremos os arquivos `menu_page_object.resource`, `account_page_object.resource` e `register_page_object.resource`. Em geral, somente é necessária a seção de `Variables` em arquivos de *Page Objects*, porém, é possível e até recomendada a utilização do comando `Documentation` da seção `Settings` em todos os arquivos para melhor documentação do projeto.

Mapeamentos de objetos do menu:

```
*** Variables ***
${MY_ACCOUNT_BUTTON}    Minha conta
```

Mapeamentos de objetos da página "Minha conta":

```
*** Variables ***
${MY_WISH_LIST_LINK}  /my-wish-list
${MY_ORDERS_LINK}     /my-orders
${MY_INVOICES_LINK}   /my-invoices
```

E por fim, os mapeamentos de objetos do cadastro:

```
*** Variables ***
${NAME_TEXT_FIELD}           id: register-name
${MAIL_TEXT_FIELD}           id: register-mail
${PASS_TEXT_FIELD}           id: register-pass
${CONFIRM_PASS_TEXT_FIELD}   id: register-confirm-pass
${BUSINESS_ENTITY_COMBOBOX}  id: register-user-type
${TERMS_OF_USE_CHECKBOX}     id: terms-of-use
${REGISTER_BUTTON}           name: register
```

Após a criação dos novos arquivos, é necessário importá-los no caso de teste. Isso é feito através do comando `Resource`, na seção `Settings`, conforme o trecho de código a seguir:

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.

Library         SeleniumLibrary
Resource        config.resource
Resource        menu_page_object.resource
Resource        account_page_object.resource
Resource        register_page_object.resource

Test Setup      Open Browser   ${URL}  ${BROWSER}
Test Teardown   Close Browser
```

Agora que temos vários arquivos em nosso projeto, precisamos organizá-los em diretórios para que não se torne um projeto confuso. Seguindo os padrões da tecnologia, devemos inserir todas as suítes de teste no diretório `tests/` e todos os arquivos auxiliares no diretório `resources/`. Criaremos ainda um subdiretório específico para os mapeamentos de objetos, chamado `page_objects/`. A árvore de diretórios do projeto ficará assim:

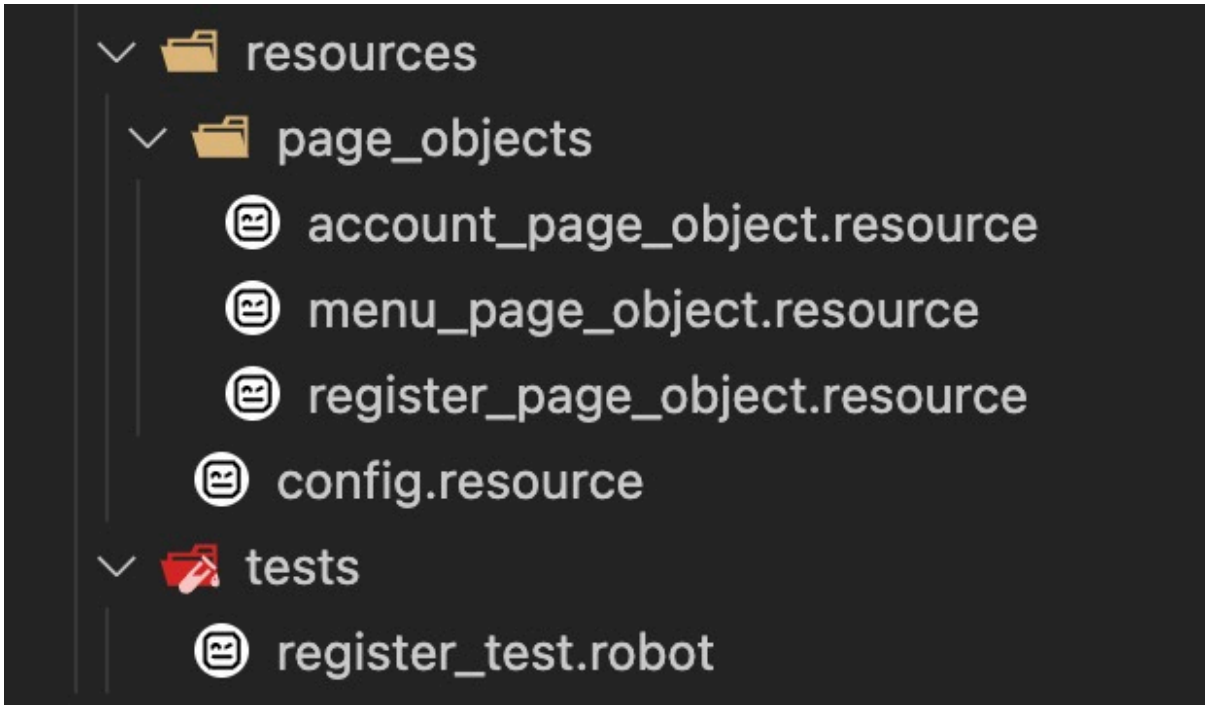


Figura 3.1: Organização do projeto após isolamento dos mapeamentos de objetos.

Não se esqueça de atualizar os comandos de importação dos arquivos auxiliares.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.

Library         SeleniumLibrary

Resource        ../resources/config.resource
Resource        ../resources/page_objects/menu_page_object.resource
Resource        ../resources/page_objects/account_page_object.resource
Resource        ../resources/page_objects/register_page_object.resource

Test Setup      Open Browser   ${URL}  ${BROWSER}
Test Teardown   Close Browser
```

De agora em diante, o comando de execução referenciará o diretório `tests/` em vez do arquivo de suíte especificamente:

```
$ robot -d reports/ tests/
```

3.3 Abstração das ações do sistema

Retomaremos agora a descrição do cenário de testes que criamos para cadastro de pessoa física no capítulo anterior. Nosso script executa os passos corretamente, mas ainda não está tão legível quanto poderia. Podemos implementar nosso teste de forma que fique muito parecido com o que foi planejado.

```
Como cliente da loja virtual, desejo criar um cadastro no sistema para
poder visualizar meus pedidos e lista de desejos.

Cenário: Cadastro com sucesso de Pessoa Física
    Dado que estou na página da loja virtual
      E acessei a área de cadastro
  Quando preencho o formulário com meus dados
      E submeto os dados do formulário
      E aceito os termos de uso
   Então devo receber uma mensagem de sucesso
      E devo visualizar a opção de acessar meus pedidos
      E devo visualizar a opção de acessar minha lista de desejos
```

Podemos abstrair os passos implementados em comandos chamados *keywords* que descrevam com mais precisão seus objetivos de negócio. As *keywords* podem ser abstraídas dentro de outras *keywords* sem limites, e o ponto de equilíbrio é uma questão de modelagem da solução.

As *keywords* customizadas devem ser escritas na seção de `Keywords` . Reescreveremos o cenário de cadastro de pessoa física conforme seria ideal com base na descrição que fizemos em Gherkin, traduzido para o idioma inglês para manter a harmonia com os comandos do Robot e seguir o padrão da maioria das empresas do mercado. A

sinalização `[...]` indica que trechos de código foram omitidos do exemplo pois não são relevantes neste momento.

```
[...]

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill register form
      And I accept the terms of use
      And I submit register form
     Then I should see a success message
      And I should see My Wish List
      And I should see My Orders

[...]

*** Keywords ***
I am on Bookstore Demo web site
  Title Should Be  Bookstore Demo

I go to log in or sign in page
  Click Link     ${MY_ACCOUNT_BUTTON}

I fill register form
  Input Text                 ${NAME_TEXT_FIELD}           Lorem ipsum
  Input Text                 ${MAIL_TEXT_FIELD}           lorem@ipsum.com
  Input Password             ${PASS_TEXT_FIELD}           foo123
  Input Password             ${CONFIRM_PASS_TEXT_FIELD}   foo123
  Select From List By Label  ${BUSINESS_ENTITY_COMBOBOX}  Pessoa Física

I accept the terms of use
  Select Checkbox            ${TERMS_OF_USE_CHECKBOX}

I submit register form
  Submit Form     ${REGISTER_BUTTON}

I should see a success message
```

```
  Wait Until Page Contains   Lorem ipsum, teu cadastro foi realizado com
sucesso!

I should see My Orders
  Page Should Contain Link   ${MY_ORDERS_LINK}

I should see My Wish List
  Page Should Contain Link   ${MY_WISH_LIST_LINK}
```

> Quando houver conflito de *keywords* com o mesmo nome, o Robot Framework definirá a prioridade baseado no escopo de declaração dessas *keywords*, com base na seguinte ordem:
>
> 1. *Keywords* declaradas no próprio arquivo de casos de teste;
> 2. *Keywords* declaradas em arquivos *.resource*, importados direta ou indiretamente;
> 3. *Keywords* de bibliotecas externas, como o Selenium, por exemplo;
> 4. *Keywords* da biblioteca padrão do Robot Framework.

Assim como os mapeamentos dos objetos foram encapsuladas em arquivos próprios com o objetivo de organização e reutilização, faremos o mesmo com as *keywords*. Estamos falando da camada intermediária da arquitetura de Page Objects, onde serão abstraídas as **ações** do sistema. Devemos criar um arquivo para cada funcionalidade, agrupando as ações de cada uma delas, que, por sua vez, interagirão com os objetos da tela. Esses arquivos devem estar em um subdiretório chamado `pages/`, dentro do diretório `resources/`. Separaremos nossas *keywords* em `home.resource`, `menu.resource`, `register.resource` e `account.resource`.

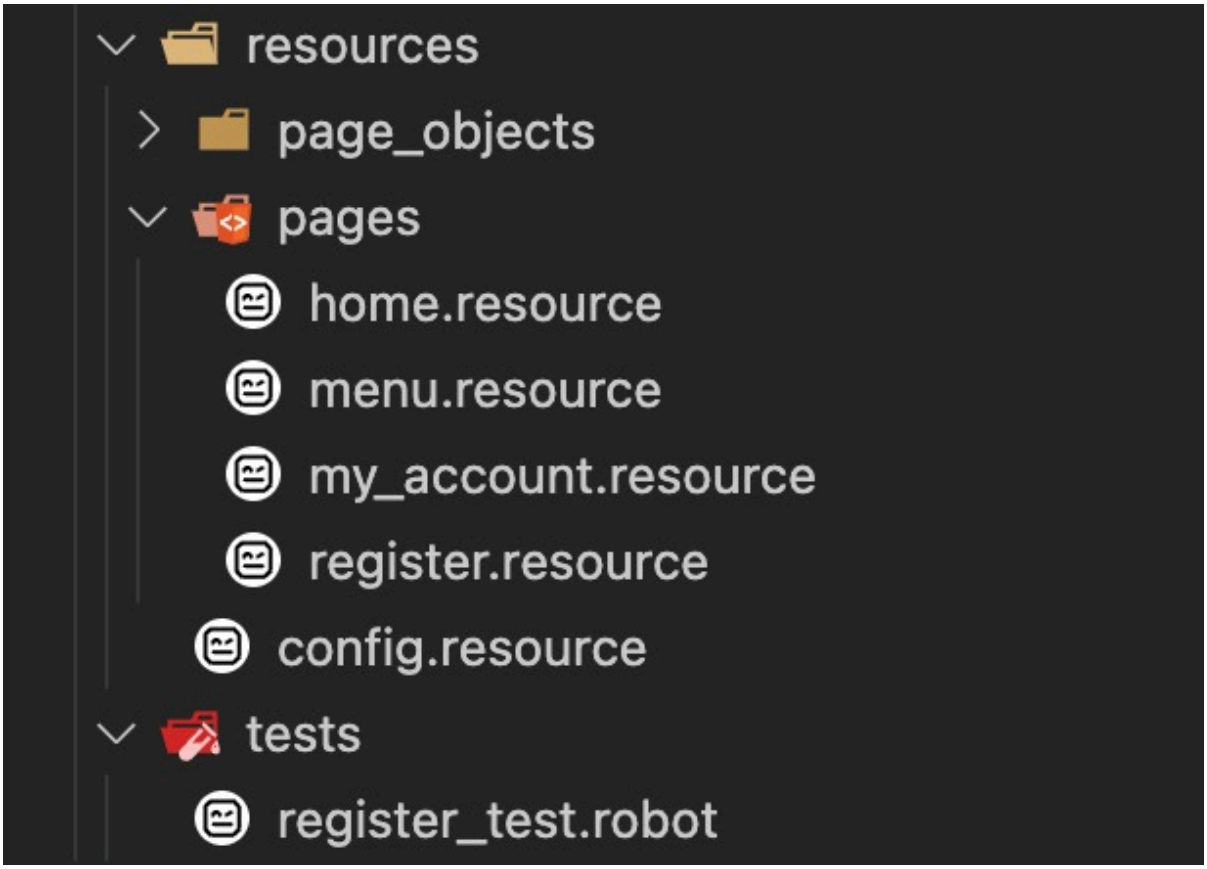


Figura 3.2: Organização do projeto após abstração das ações do sistema.

Lembre-se de que a importação dos recursos de *Page Objects* deve ser movida para estes novos arquivos, e substituída nas suítes de teste pelos arquivos da nova camada que acabamos de criar, denominada de *tasks* ou outros termos equivalentes como *actions*, por exemplo.

Assim como os *Page Objects*, os arquivos da camada de *tasks* também podem — e devem — possuir o comando `Documentation` em sua seção *Settings*. A seguir, o arquivo `register.resource`.

```
*** Settings ***
Library   SeleniumLibrary
Resource  ../page_objects/register_page_object.resource

*** Keywords ***
I fill register form
```

```
  Input Text                 ${NAME_TEXT_FIELD}           Lorem ipsum
  Input Text                 ${MAIL_TEXT_FIELD}           lorem@ipsum.com
  Input Password             ${PASS_TEXT_FIELD}           foo123
  Input Password             ${CONFIRM_PASS_TEXT_FIELD}   foo123
  Select From List By Label  ${BUSINESS_ENTITY_COMBOBOX}  Pessoa Física

I accept the terms of use
  Select Checkbox            ${TERMS_OF_USE_CHECKBOX}

I submit register form
  Submit Form                ${REGISTER_BUTTON}

I should see a success message
  Wait Until Page Contains   Lorem ipsum, teu cadastro foi realizado com
sucesso!
```

A granularidade de cada ação é uma decisão subjetiva, de forma que um testador pode construir um comando como "cadastrar novo produto no sistema"; outros podem implementar o mesmo teste utilizando comandos como "acessar tela de cadastro de produto", "preencher detalhes do produto", "inserir imagens do produto" e "salvar dados do novo produto"; há ainda quem prefira escrever os cenários de teste com passos mais atômicos, como "clicar no botão 'Novo produto'", "esperar a tela de cadastro carregar", "inserir o texto XXX no campo de texto 'Nome do produto'", assim por diante. Todas as formas são válidas.

O Robot nos permite criar infinitas *keywords* que agrupam *keywords*, de modo que é possível criar mais de uma camada de ações do sistema, mas isso é uma decisão de projeto. Recomendo refletir sobre como escrever os casos de teste de forma mais compreensível possível em nível de negócio, e sobre como tornar o projeto mais fácil de dar manutenção como um todo.

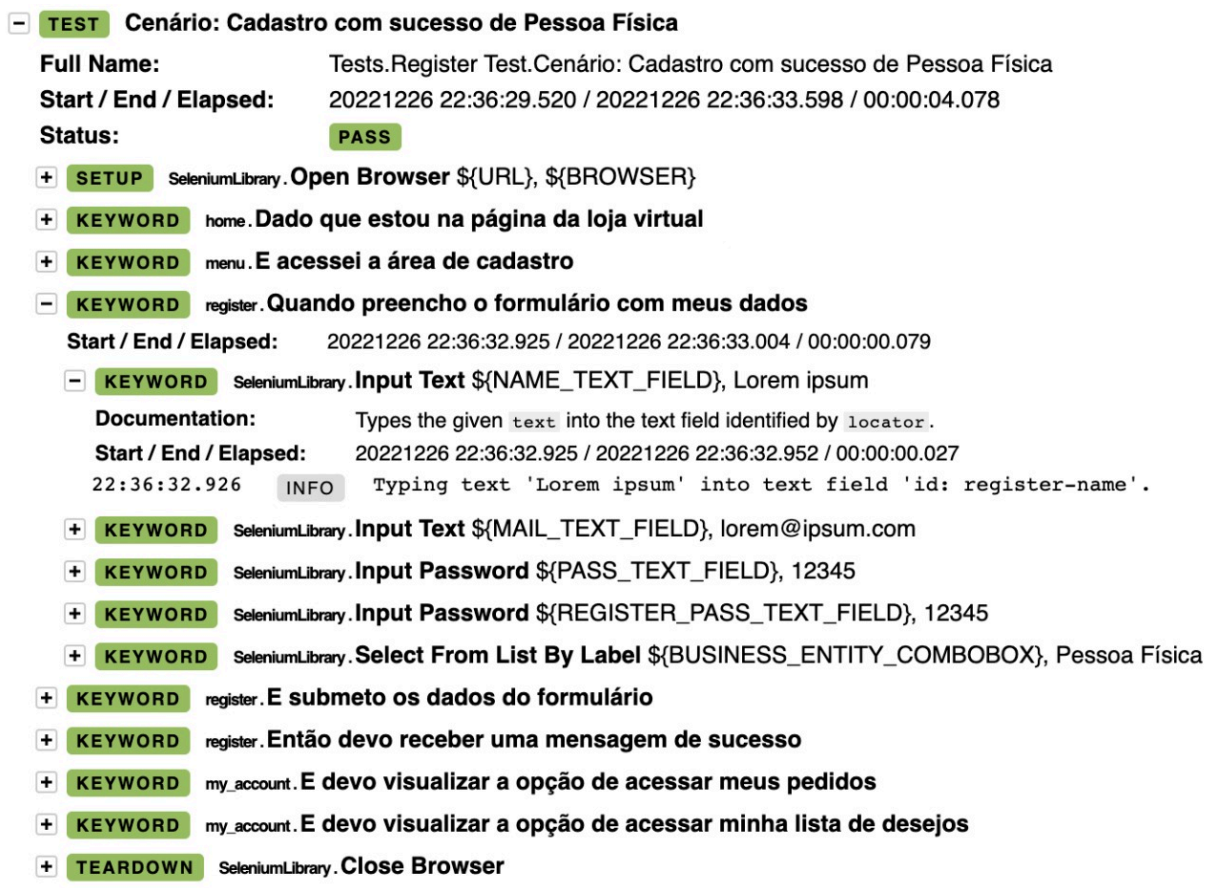


Figura 3.3: O relatório exibe detalhadamente a hierarquia de keywords.

Em geral, um nome consistente para uma *keyword* indica apenas um único objetivo, como "buscar produto" e "adicionar produto ao carrinho", e nunca "buscar produto e adicionar ao carrinho". O que é fundamental é que as *keywords* estejam preparadas para serem reutilizadas, mas falaremos mais sobre esse aspecto no próximo capítulo.

A partir deste momento, nosso projeto terá três camadas de abstração: objetos, ações e descrição dos cenários de teste. No anexo II deste volume você encontrará um artigo aprofundado sobre abstração de camadas em projetos de automação com padrão de *Page Objects*.

Definições de idioma

Desde a versão 6.0 do framework, é possível trabalhar com outros idiomas além do inglês, que é o idioma padrão. Ao definir outro idioma de trabalho, todos os comandos e seções podem ser traduzidos, mas o que nos interessa aqui é utilizar as diretivas da sintaxe Gherkin *Dado/Quando/Então* em vez de *Given/When/Then* para que os casos de testes sejam escritos em português. Os comandos das bibliotecas como o Selenium seguem em inglês, a menos que os desenvolvedores dessas bibliotecas disponibilizem os comandos traduzidos. Para habilitar o idioma português brasileiro, devemos inserir o comando `Language: PT-br` na primeira linha do arquivo de scripts.

```
Language: PT-br

*** Configurações ***
Documentação    Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.

Biblioteca      SeleniumLibrary

Recurso        ../resources/config.resource
Recurso        ../resources/pages/home.resource
Recurso        ../resources/pages/menu.resource
Recurso        ../resources/pages/my_account.resource
Recurso        ../resources/pages/register.resource

Inicialização de Teste      Open Browser   ${URL}  ${BROWSER}
Finalização de Teste        Close Browser

*** Casos de Teste ***
Cenário: Cadastro com sucesso de Pessoa Física
    Dado que estou na página da loja virtual
      E acessei a área de cadastro
  Quando preencho o formulário com meus dados
      E submeto os dados do formulário
      E aceito os temos de uso
   Então devo receber uma mensagem de sucesso
```

```
      E devo visualizar a opção de acessar meus pedidos
      E devo visualizar a opção de acessar minha lista de desejos

Cenário: Cadastro com sucesso de Pessoa Jurídica
    [...]
```

Na camada de *tasks*, somente o nome das *keywords* são traduzidos. O arquivo `register.resource` atualizado ficaria:

```
*** Settings ***
Library   SeleniumLibrary
Resource  ../page_objects/register_page_object.resource

*** Keywords ***
preencho o formulário com meus dados
  Input Text                 ${NAME_TEXT_FIELD}           Lorem ipsum
  Input Text                 ${MAIL_TEXT_FIELD}           lorem@ipsum.com
  Input Password             ${PASS_TEXT_FIELD}           foo123
  Input Password             ${CONFIRM_PASS_TEXT_FIELD}   foo123
  Select From List By Label  ${BUSINESS_ENTITY_COMBOBOX}  Pessoa Física

aceito os termos de uso
  Select Checkbox            ${TERMS_OF_USE_CHECKBOX}

submeto os dados do formulário
  Submit Form                ${REGISTER_BUTTON}

devo receber uma mensagem de sucesso
  Wait Until Page Contains   Lorem ipsum, teu cadastro foi realizado com
sucesso!
```

Recomendo cuidado e organização para não misturar dois idiomas e tornar o código confuso. Em geral, a escolha do idioma dos casos de teste dependerá do contexto do time e da empresa, pois representam a camada mais abstrata e devem ser compreensíveis para todas as pessoas envolvidas. Por outro lado, recomendo manter em inglês questões de implementação em outras camadas do projeto, como nomes de arquivos e variáveis, visto que os padrões e conceitos implementados universalmente utilizam termos em inglês.

Cada empresa terá sua política interna de padronização de idioma para escrita dos scripts. Apesar de este livro ser concebido para um público cujo idioma nativo é o português brasileiro, assumiremos que o inglês é o idioma adotado pela empresa para desenvolvimento. A funcionalidade de tradução dos scripts Robot é recente, disponibilizada a partir da versão 6.0 do framework. Não tive a oportunidade de conhecer algum projeto que tenha adotado um idioma diferente do inglês para desenvolvimento dos testes.

Para visualizar todos os idiomas disponíveis e a tradução de cada comando existente, consulte: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#enabling-languages.

3.4 Parâmetros e argumentos

Agora que nosso cenário de teste de cadastro de pessoa física está refatorado, faremos o mesmo no cenário de teste seguinte, de cadastro de pessoa jurídica. Substituiremos os comandos básicos do Selenium por *keywords* personalizadas que tornarão o passo a passo mais legível.

```
[...]

Scenario: Success sign in as Juridical Person
    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill register form
      And I submit register form
      And I accept the terms of use
     Then I should see a success message
      And I should see My Invoices
      And I should see My Orders
```

Perceba que os passos para cadastro de pessoa física e jurídica são exatamente os mesmos, e a diferença essencial são os dados utilizados no preenchimento dos campos do formulário. Isso significa que não há motivos para criarmos novas *keywords* para realizar exatamente as mesmas ações no sistema, mas precisamos realizar as mesmas ações com dados diferentes. O que deve ser feito é reorganizar estas ações para que os dados utilizados sejam parametrizáveis. Não deve haver dados de teste na implementação dos comandos da camada de *tasks*. Os dados devem ser abstraídos em outro componente para que as ações sejam executadas independentemente deles. Dessa maneira, os comandos poderão ser reaproveitados e o código ficará mais limpo e coeso.

Assim como métodos e funções em outras linguagens de programação permitem o envio de parâmetros quando invocadas, as *user keywords* do Robot Framework também. Esta talvez seja a parte mais complicada da sintaxe do Robot, devido às variadas formas de definir parâmetros para as *keywords*. Os dois principais tipos de parâmetros são os **argumentos posicionais** (*Positional arguments*) e os **argumentos embutidos** (*Embedded arguments*). Não é permitido utilizar ambos em uma mesma *keyword*.

Argumentos posicionais

Os argumentos posicionais são os mais simples e mais utilizados. Devemos listar as variáveis que receberão os parâmetros através do

comando `[Arguments]`, na primeira linha do código de definição da *user keyword*. Quando esta *keyword* for invocada, basta inserirmos os argumentos na ordem correta, ou informar os nomes dos argumentos explicitamente como *named arguments*. A maioria dos comandos nativos do Robot e do Selenium funcionam dessa forma.

Neste momento, os comandos `I am on Bookstore Demo web site` e `I go to log in or sign in page` não precisarão de parametrização, mas para o comando `I fill register form` deveremos informar os dados nome, e-mail, senha, confirmação de senha e tipo de pessoa. O comando `I should see a success message` deve receber a mensagem que o teste espera que seja exibida no sistema. Implementando esses comandos com parâmetros posicionais, as *keywords* em `register.resource` ficariam assim:

```
[...]

*** Keywords ***
I fill register form
  [Arguments]     ${name}  ${mail}  ${pass}
          ...     ${confirm_pass}   ${business_entity}
  Input Text      ${NAME_TEXT_FIELD}           ${name}
  Input Text      ${MAIL_TEXT_FIELD}           ${mail}
  Input Password  ${PASS_TEXT_FIELD}           ${pass}
  Input Password  ${CONFIRM_PASS_TEXT_FIELD}   ${confirm_pass}
  Select From List By Label  ${BUSINESS_ENTITY_COMBOBOX}
${business_entity}

I should see a success message
  [Arguments]  ${message}
  Wait Until Page Contains   ${message}

[...]
```

O comando `...` na linha seguinte à configuração `[Arguments]` serve apenas para quebrar a linha, tornando a listagem de variáveis mais legível, e não possui nenhuma interferência prática na execução do

script. A utilização da palavra-chave parametrizada no caso de teste, no arquivo `register_test.robot`, fica como no exemplo a seguir:

```
[...]

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill register form
          ...  name=Lorem ipsum  mail=lorem@ipsum.com  pass=foo123
          ...  confirm_pass=foo123  business_entity=Pessoa Física
      And I accept the terms of use
      And I submit register form
     Then I should see a success message  message=Lorem ipsum, teu
cadastro foi realizado com sucesso!
      And I should see My Wish List
      And I should see My Orders

Scenario: Success sign in as Juridical Person
    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill register form
          ...  name=Lorem ipsum LTDA    mail=contact@ipsum.com
          ...  pass=foo123    confirm_pass=foo123
          ...  business_entity=Pessoa Jurídica
      And I accept the terms of use
      And I submit register form
     Then I should see a success message  message=Lorem ipsum LTDA, teu
cadastro foi realizado com sucesso!
      And I should see My Invoices
      And I should see My Orders

[...]
```

Assim como na definição da *keyword*, no caso de teste os trechos com `...` servem apenas para quebrar a linha, sem modificar o comportamento do comando. Devemos observar, justamente, que a necessidade de tal procedimento nos indica que a quantidade de

parâmetros ficou grande e que talvez exista outra forma melhor de implementar este trecho do caso de teste, tornando-o mais simples e legível. Vejamos como ficaria a implementação com os argumentos embutidos.

Argumentos embutidos

Os argumentos embutidos permitem a passagem de argumentos diretamente no nome da *keyword* em vez de listá-los nas cédulas ao lado do nome da palavra-chave. Esse tipo de abordagem pode tornar o código mais legível e compreensível mesmo para quem não possui conhecimentos em Robot Framework. No entanto, nessa abordagem não é possível utilizar valores padrões ou quantidade variável de argumentos.

Em nossa implementação do comando de preenchimento do formulário, percebemos que a quantidade de argumentos necessários para a ação ficou maior do que gostaríamos, então aproveitaremos os argumentos embutidos para refatorar o código da palavra-chave e dividi-la em ações menores, mais fáceis de ler e manter.

Perceba que a utilização das variáveis que recebem os valores dos parâmetros não muda. A grande diferença está na forma como são declaradas e utilizadas, neste caso, no meio do próprio nome da *keyword*. **O uso das aspas não é necessário**: elas estão sendo utilizadas neste caso para facilitar a compreensão dos exemplos. A seguir, veja o código do arquivo `register.resource` refatorado:

```
[...]

*** Keywords ***
I fill "${name}" in name text field
  Input Text      ${NAME_TEXT_FIELD}          ${name}

I fill "${mail}" in e-mail text field
```

```
  Input Text      ${MAIL_TEXT_FIELD}          ${mail}

I fill "${pass}" in pass text field
  Input Text      ${PASS_TEXT_FIELD}          ${pass}

I fill "${confirm_pass}" in confirm pass text field
  Input Text      ${CONFIRM_PASS_TEXT_FIELD}   ${confirm_pass}

I select "${business_entity}" in business entity combo box
  Select From List By Label  ${BUSINESS_ENTITY_COMBOBOX}
${business_entity}

I should see the "${message}" message
  Wait Until Page Contains   ${message}

[...]
```

Na invocação dos novos comandos, no arquivo `register_test.robot`, apesar de a quantidade de linhas ter aumentado, a legibilidade também aumentou.

```
[...]

*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill "Lorem ipsum" in name text field
      And I fill "lorem@ipsum.com" in e-mail text field
      And I fill "foo123" in pass text field
      And I fill "foo123" in confirm pass text field
      And I select "Pessoa Física" in business entity combo box
      And I accept the terms of use
      And I submit register form
     Then I should see the "Lorem ipsum, teu cadastro foi realizado com
sucesso!" message
      And I should see My Wish List
      And I should see My Orders
```

```
Scenario: Success sign in as Juridical Person
    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill "Lorem ipsum LTDA" in name text field
      And I fill "contact@loremipsum.com" in e-mail text field
      And I fill "foo123" in pass text field
      And I fill "foo123" in confirm pass text field
      And I select "Pessoa Jurídica" in business entity combo box
      And I accept the terms of use
      And I submit register form
     Then I should see the "Lorem ipsum LTDA, teu cadastro foi realizado
com sucesso!" message
      And I should see My Invoices
      And I should see My Orders

[...]
```

Em nosso caso, a abordagem mais adequada é a que utiliza parâmetros embutidos, mas essa decisão dependerá sempre do contexto. Em outros casos do mesmo projeto, pode ser mais sensato optar pela implementação com parâmetros posicionais. Um forte motivo para essa escolha é que os parâmetros embutidos facilitam a implementação de sentenças adequadas à sintaxe de Gherkin, a qual estamos utilizando aqui, de maneira semelhante à implementação de *step definitions* que seria feita com Cucumber (framework para escrita de testes com Gherkin). Cabe à pessoa desenvolvedora analisar caso a caso e modelar a implementação do projeto visando melhor legibilidade e manutenibilidade.

Valores padrões para argumentos

Enquanto trabalhamos com argumentos posicionais, se não definirmos um valor padrão para um argumento, automaticamente ele será considerado obrigatório.

Inspirada na sintaxe do Python, a sintaxe para definição de valor padrão é `${my_arg}=default`. Neste exemplo, caso o comando seja utilizado sem que o usuário defina um valor para o argumento

`${my_arg}`, a ele será atribuído automaticamente a *string* `default` como valor padrão. Esse valor poderia ser um número, um booleano ou até mesmo outra variável que tenha sido declarada dentro do mesmo escopo ou de forma global.

Quantidade variável de argumentos

As palavras-chaves de usuário do Robot suportam quantidade variável de parâmetros posicionais. Basta utilizar uma variável do tipo lista após os parâmetros comuns. A seguir, analise o trecho de código que exemplifica a declaração de uma *user keyword* com três tipos distintos de parâmetros: um obrigatório e sem valor padrão, outro opcional com valor padrão igual a 1 e por fim uma variável do tipo lista para passagem de quantidade variável de argumentos.

```
*** Keywords ***
Required, Default, Varargs
    [Arguments]    ${req}    ${opt}=1    @{others}
    Log            Required: ${req}
    Log            Optional: ${opt}
    Log Many       @{varargs}
```

Documentação de palavras-chaves do usuário

Keywords podem receber uma breve documentação, da mesma forma que os casos de teste, através da configuração `[Documentation]`. O próprio nome das *keywords* devem ser autoexplicativos e inteligíveis, mas inserir um texto explicativo como documentação pode ser uma atitude sensata para facilitar a compreensão sobre como utilizá-las e sobre seus objetivos. Esta informação será útil não somente quando outra pessoa ler o código-fonte, como também nos relatórios de execução dos testes e na documentação automática do projeto (chamada de *Library documentation tool*, a Libdoc), que nem sempre envolverão profissionais com conhecimento técnico em Robot Framework.

A documentação pode ter múltiplas linhas. No exemplo a seguir, demonstro como utilizar um breve texto como documentação de uma palavra-chave de usuário:

```
I should see the "${message}" message
 [Documentation]  Esta palavra-chave aguardará por que a
             ...  mensagem esperada seja exibida na tela por
             ...  até 10 segundos. Caso contrário, o teste
             ...  falhará por timeout.
 Wait Until Page Contains   text: ${message}  timeout: 10
```

Para mais informações, consulte: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-name-and-documentation.

A recomendação ao criar parâmetros é utilizar nomes expressivos e descritivos, facilitando a compreensão por outras pessoas programadoras. O padrão da tecnologia é utilizar letras minúsculas para a nomenclatura de argumentos, por exemplo, `${snake_case}`, o mesmo padrão utilizado para variáveis comuns.

As palavras-chaves de usuário também podem retornar valores, assim como métodos e funções de outras linguagem são capazes. No entanto, como isso não será útil neste ponto do nosso projeto, falaremos sobre elas mais adiante. Caso deseje implementar dessa forma, o comando de retorno de valores é o `RETURN`, seguido pelo valor ou variável que se deseja retornar. Note que o comando é *case-sensitive* e deve sempre ser escrito em letras maiúsculas. A sintaxe `[Return]` ainda funciona e surte o mesmo efeito, mas foi descontinuada na versão 6.0 do Robot Framework e não deve ser utilizada.

Existem alguns recursos avançados para explorar durante a definição de parâmetros de *keywords*, como *free-named arguments* (argumentos que podem receber qualquer nome arbitrariamente definido pela pessoa programadora) e *named-only arguments* (argumentos que só podem ser utilizados de modo nomeado e não somente posicional). Para conhecer todas as possibilidades sobre parametrização de palavras-chaves do Robot, você pode conferir a documentação oficial disponível em: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#user-keyword-arguments.

CAPÍTULO 4
Lidando com a massa de dados dos testes

Os dados de teste são um fator essencial para a correta execução dos scripts, visto que não deve ser necessário que um(a) profissional prepare ou exclua os dados para o teste. O robô deve ser capaz de buscar e limpar todos dados que forem necessários, caso contrário, a suíte de testes se tornará inútil. Escrever testes com dados fixos e dependentes de contexto é um erro.

Martin apresenta, em seu livro *Clean Code*, os conceitos de F.I.R.S.T., que consistem em cinco princípios inegociáveis para construção de bons testes: cada caso de teste deve ser rápido (*Fast*), independente (*Independent*), repetível (*Repeatable*), autovalidado (*Self-validated*) e construído no tempo correto (*Timely*).

Independência se refere não somente a pré-condições de outros cenários, mas também dos dados envolvidos no teste, que deve poder ser executado em qualquer ambiente, em qualquer ordem e repetidas vezes. Por isso é tão importante pensarmos em uma estratégia consistente para a massa de dados dos testes.

A cada nova execução dos nossos cenários de teste, se não alterarmos o e-mail que está sendo cadastrado, visualizaremos a seguinte mensagem de erro no sistema, que provocará falha na execução do teste:

Figura 4.1: Mensagem de erro: e-mail já cadastrado no sistema.

4.1 Dataprovider: geração dinâmica de massa de dados

Nós precisamos gerar novos endereços de e-mail de forma dinâmica para que nosso script não falhe a cada nova execução. Felizmente, existe uma biblioteca disponível em Robot para realizar essa tarefa: a **Faker Library**. Com ela, podemos gerar dados dinâmicos, randômicos ou pseudoaleatórios (existe uma discussão conceitual sobre a capacidade de uma máquina comum gerar dados realmente aleatórios), tais como números simples, números de documentos, números de telefone, nomes de pessoas, endereços, dentre muitos outros.

A biblioteca do Faker não vem por padrão junto ao Robot Framework, portanto, é necessário executar um comando para

realizar a instalação através do PIP.

```
$ pip install robotframework-faker
```

Em seguida, é preciso importar a biblioteca na seção de configurações do arquivo onde ela será utilizada, da mesma forma que fizemos com a biblioteca do próprio Selenium, utilizando o comando `Library`.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.

Library         FakerLibrary

[...]
```

Devemos criar uma variável para armazenar o valor gerado pelo Faker e utilizá-la no lugar onde antes estava o dado fixo no código. Preste atenção ao nome que será dado à variável para que fique claro que o valor que ela guardará é um dado *fake*. Para não afetar a legibilidade do cenário de teste, sugiro separar todos os comandos de geração de dados antes dos comandos principais do script.

```
[...]

*** Test Cases ***
Scenario: Success sign in as Natural Person
  ${fake_email} =  FakerLibrary.Email

    Given I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill "Lorem ipsum" in name text field
      And I fill "${fake_email}" in e-mail text field
      And I fill "foo123" in pass text field
      And I fill "foo123" in confirm pass text field
      And I select "Pessoa Física" in business entity combo box
      And I submit register form
     Then I should see the "Lorem ipsum, teu cadastro foi realizado com
```

```
sucesso!" message
      And I should see My Wish List
      And I should see My Orders

[...]
```

Após a execução, é possível verificar nos logs quais dados foram gerados pelo Faker.

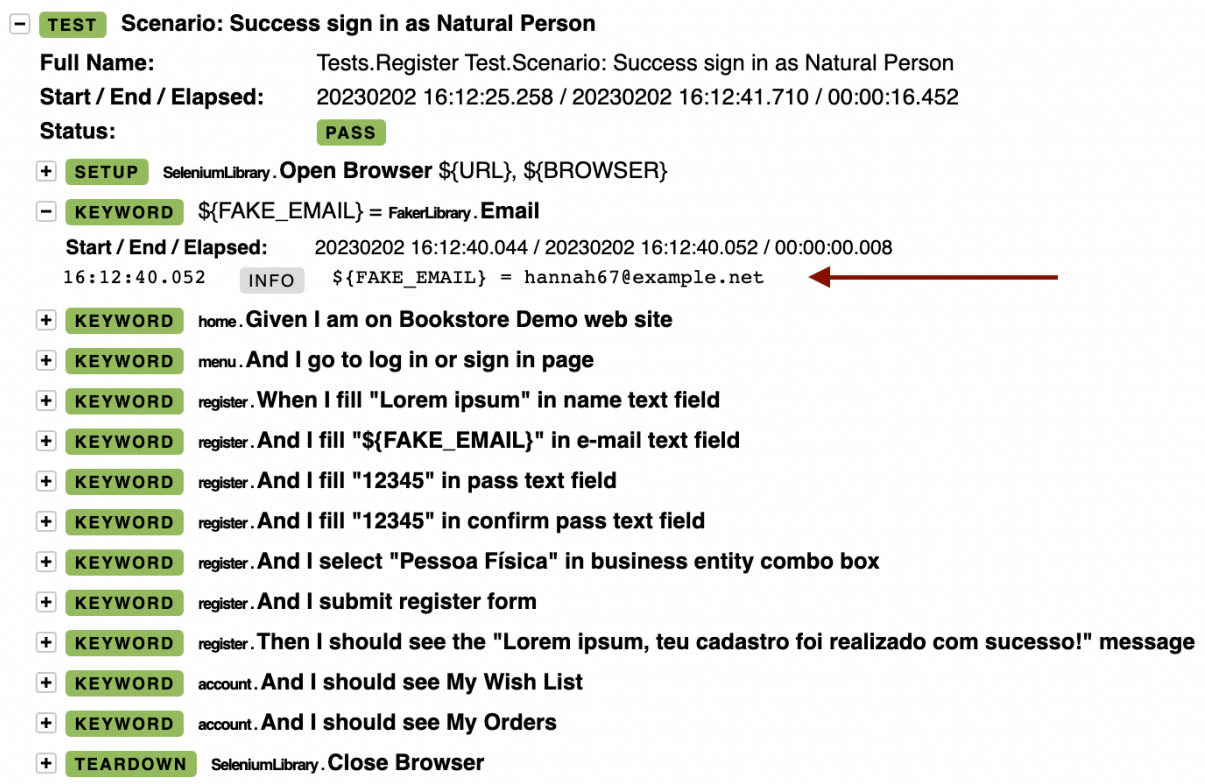


Figura 4.2: Logs da execução exibirão os dados gerados pelo Faker.

Dessa forma, o script pode parecer não tão amigável e facilmente legível para não programadores, especialmente se a quantidade de dados gerados aumentar. Existe uma estratégia para abstrair o gerador de dados utilizando variáveis globais, no entanto, é necessário ter bastante atenção ao trabalhar dessa forma, caso contrário, comportamentos indesejáveis podem ocorrer. Vejamos como isso funciona.

Criaremos um novo arquivo no diretório `resources/` que será responsável pela geração de todos os dados dinâmicos do projeto, chamado `dataprovider.resource`. Podemos transferir a importação da biblioteca do Faker Library para este novo arquivo, pois de agora em diante somente este arquivo precisará interagir com os comandos do Faker.

Após a geração do dado, poderíamos utilizar o recurso de retornar o valor através do comando `RETURN`, de forma análoga ao comportamento de funções em outras linguagens de programação comuns.

```
*** Settings ***
Library         FakerLibrary

*** Keywords ***
I generated a fake e-mail
  ${fake_email} =  FakerLibrary.Email
  RETURN   ${fake_email}
```

No script do cenário de testes, devemos importar o arquivo recém-criado e invocar a nova palavra-chave criada com a diretiva `Given` do Gherkin, onde devem estar os passos de pré-condição do teste.

Contudo, com esta abordagem, o script do cenário de teste seguirá um pouco complicado:

```
[...]

*** Test Cases ***
Scenario: Success sign in as Natural Person
  ${fake_email} =  Given I generated a fake e-mail
                     And I am on Bookstore Demo web site
                     And I go to log in or sign in page
                    When I fill "Lorem ipsum" in name text field
                     And I fill "${fake_email}" in e-mail text field
                     And I fill "foo123" in pass text field
                     And I fill "foo123" in confirm pass text field
                     And I select "Pessoa Física" in business entity combo
```

```
box
                    And I submit register form
                   Then I should see the "Lorem ipsum, teu cadastro foi
realizado com sucesso!" message
                    And I should see My Wish List
                    And I should see My Orders

[...]
```

Para que o script fique o mais simples possível, removeremos a atribuição à variável `${fake_email}` do cenário de teste, passando essa atribuição para o componente de Dataprovider e utilizando o recurso de variável de teste.

As variáveis de teste são definidas através da palavra-chave `Set Test Variable` da biblioteca padrão do Robot. Os dados armazenados em variáveis de teste terão como escopo a execução do caso de teste. Isso quer dizer que a variável não será compartilhada entre casos de teste nem será acessível em escopo Global ou de Suíte, mas, sim, em todos os pontos do código-fonte que estivem dentro de determinado caso de teste. Por ser uma variável que pode não ter sido declarada no mesmo arquivo no qual é utilizada, a recomendação é seguir o padrão de nomenclatura das variáveis globais, em ALL_CAPS, ou seja, `${FAKE_EMAIL}` .

```
*** Settings ***
Library         FakerLibrary

*** Keywords ***
I generated a fake e-mail
  ${fake_email} =    FakerLibrary.Email
  Set Test Variable  ${FAKE_EMAIL}  ${fake_email}
```

Assim podemos escrever outra palavra-chave específica para preenchimento do campo *e-mail* com o valor randômico gerado pelo Faker, para que o cenário de teste fique livre de detalhes técnicos de implementação:

```
[...]

I fill the fake e-mail in e-mail text field
  Input Text                ${MAIL_TEXT_FIELD}          ${FAKE_EMAIL}
```

E o cenário de teste ficará novamente simples de ler, com a geração dinâmica dos dados completamente abstraída. É muito importante ter atenção ao trabalhar com essa abordagem, pois, se esquecermos de utilizar a palavra-chave `I generated a fake e-mail`, o conteúdo da variável `${FAKE_EMAIL}` será imprevisível, podendo tanto estar vazia quanto conter um dado desconhecido.

```
*** Test Cases ***
Scenario: Success sign in as Natural Person
    Given I generated a fake e-mail
      And I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill "Lorem ipsum" in name text field
      And I fill the fake e-mail in e-mail text field
      And I fill "foo123" in pass text field
      And I fill "foo123" in confirm pass text field
      And I select "Pessoa Física" in business entity combo box
      And I submit register form
     Then I should see the "Lorem ipsum, teu cadastro foi realizado com
sucesso!" message
      And I should see My Wish List
      And I should see My Orders

Scenario: Success sign in as Juridical Person
    Given I generated a fake e-mail
      And I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill "Lorem ipsum LTDA" in name text field
      And I fill the fake e-mail in e-mail text field
      And I fill "foo123" in pass text field
      And I fill "foo123" in confirm pass text field
      And I select "Pessoa Jurídica" in business entity combo box
      And I submit register form
     Then I should see the "Lorem ipsum LTDA, teu cadastro foi realizado
```

```
com sucesso!" message
      And I should see My Invoices
      And I should see My Orders
```

Agora nosso script funcionará como esperado. Em alguns casos, será interessante gerar os outros dados dinamicamente também, como nome e senha, pois variar os dados utilizados ajudam a provocar falhas no sistema e tornam o teste menos "viciado". Para preservar a simplicidade de nosso exemplo, manteremos apenas a geração dos e-mails.

A documentação completa da Faker Library, onde é possível visualizar todos os tipos de dados que a biblioteca é capaz de gerar, está em https://guykisel.github.io/robotframework-faker/.

4.2 Datapool: conjunto estático de massa de dados

Datapool é um conjunto de dados estáticos configurados previamente pelo testador e que serão utilizados como entrada para os cenários de teste que precisam de determinados tipos de dados para que seja possível cobrir todas as regras de negócio da funcionalidade. Em alguns projetos, esse conjunto de dados é preparado com arquivos JSON, CSV e até mesmo TXT. No entanto, o Robot Framework disponibiliza uma forma distinta de implementação para essa forma de trabalhar: os *test templates*, que são muito parecidos com a especificação por exemplos da sintaxe de Gherkin, como está descrita em https://cucumber.io/docs/gherkin/reference/#examples.

Quando um conjunto de testes depende de variação de dados para definição dos cenários que cobrirão os requisitos testados, é possível mudarmos a abordagem de escrita dos scripts Robot — em vez de

descrever as histórias de usuário utilizando abordagem dirigida por comportamento, ou seja, *behavior-driven* ou *keyword-driven*, como fizemos até agora, podemos escrever com abordagem dirigida por dados, ou seja, *data-driven*.

Ao utilizar o recurso de templates, adotamos a abordagem dirigida por dados, e em vez de repetir múltiplas vezes a mesma sequência de ações em diferentes cenários de testes, utilizaremos apenas a definição de massa de dados como guia.

Keyword-driven style

O estilo de escrita guiado por palavras-chave é a forma mais simples de utilização do Robot Framework. São os testes descritos através de sequências de passos que interagem com diferentes elementos do sistema. Cada passo executará uma ação que, inicialmente, preparará o estado inicial do sistema, realizará alguma tarefa e então validará o estado final do sistema. Para um fluxo de login, por exemplo, o robô acessa o sistema na página de login, insere as credenciais no campo "E-mail" e no campo "Senha", clica no botão "Entrar" e, por fim, valida que é possível acessar as funcionalidades restritas do sistema.

Behavior-driven style

Estilo de escrita guiado por comportamento é a abordagem na qual os testes são descritos através de histórias de usuário ou especificação por exemplos, de maneira que profissionais das áreas de negócio possam compreender os scripts com clareza. Este aspecto é importante para os testes de aceitação que estamos escrevendo. Neste caso, normalmente utilizamos a sintaxe de Gherkin, mas não somente. Em resumo, as pré-condições serão definidas na diretiva `Dado` ; as ações, na diretiva `Quando` ; e a validação do estado final do sistema, na diretiva `Então` . A escrita dos testes deve seguir uma abordagem com maior nível de abstração.

Reescrevendo o fluxo de login, *Dado* que acessei o sistema *E* estou na página de login, *Quando* insiro minhas credenciais *Então* devo conseguir acessar as funcionalidades restritas.

A documentação completa da sintaxe de Gherkin está em https://cucumber.io/docs/gherkin/.

Data-driven style

Por fim, temos o estilo de escrita guiado por dados. Nessa abordagem, os dados são mais importantes do que o fluxo de ações e a variação destes dados é que descreverão cada cenário de teste distinto, enquanto a sequência de passos executados será sempre igual. Podemos validar, por exemplo, um caso de teste de tentativa de login com nome de usuário inválido e outro caso de teste de tentativa de login com senha inválida utilizando a mesma sequência de ações com dados diferentes.

Test templates

Test templates é o recurso do Robot Framework que deve ser utilizado para implementação de casos de teste com abordagem *data-driven*. Através dele, podemos reaproveitar um cenário de testes diversas vezes através da variação de dados de entrada, que serão definidos em formato de tabela na descrição da suíte de testes. Esse recurso pode ser útil nos próximos testes que construiremos agora. Precisamos garantir que as regras de validação do formulário de cadastro do site estão funcionando conforme o esperado. Já testamos dois cenários de sucesso, agora exploraremos alguns cenários de falha. As regras de negócio que a validação do formulário deve seguir e as quais devemos testar são bem simples:

- O campo "nome" não deve ficar vazio;
- O campo "e-mail" deve receber um endereço de correio eletrônico válido;
- O campo "senha" deve conter letras e números;
- O campo "senha" deve possuir ao menos cinco dígitos;

- O campo "confirmar senha" deve ser correspondente ao valor do campo "senha".

Como se trata da mesma funcionalidade, poderíamos aproveitar o mesmo arquivo, mas, por questão de melhor organização, criaremos um novo arquivo para nossos novos cenários de testes, chamado `register_form_validations.robot`. Recomendo não misturar diferentes abordagens de escrita de cenários de teste no mesmo arquivo.

Na seção *Settings* desse arquivo também inseriremos uma breve documentação sobre a suíte de testes, importaremos a biblioteca do Selenium, o arquivo de configurações e os *resources* que utilizaremos, e configuraremos o *test setup* e o *test teardown* exatamente como fizemos no arquivo anterior. Adicionalmente, utilizaremos o comando `Test template` para informar qual é a palavra-chave que será utilizada por padrão com a passagem de dados da tabela que construiremos na seção de casos de testes.

É essa palavra-chave que servirá de template e agrupará todo o fluxo padrão que será executado repetidas vezes com os dados da tabela. A recomendação é que essa palavra-chave seja implementada na seção *Keywords* da própria suíte. Os dados serão parametrizados através de argumentos não embutidos, como aprendemos no capítulo anterior.

Cada linha da seção *Test Cases* representará um novo caso de teste, com dados diferentes, que devem ser tabelados na sequência da linha. Escreveremos então seis linhas, para cobrir as regras de negócio de validação do nosso formulário com seis casos de teste. Também devemos informar qual a mensagem esperada do sistema, para que a validação seja realizada conforme os dados utilizados.

É importante ter em mente que a ordem das colunas da tabela deve ser a mesma dos parâmetros posicionais declarados na palavra-chave que serve de template.

```
*** Settings ***
Documentation   Como cliente da loja virtual, devo ser induzido a
```

```
           ...  criar uma senha forte ao me cadastrar no sistema.
Library         SeleniumLibrary
Resource        ../resources/config.resource
Resource        ../resources/pages/home.resource
Resource        ../resources/pages/menu.resource
Resource        ../resources/pages/register.resource

Test Setup      Open Browser   ${URL}  ${BROWSER}
Test Teardown   Close Browser
Test Template   Fill register form with data

*** Test Cases ***                    NAME         E-MAIL         PASS
CONFIRM PASS  EXPECTED MESSAGE
Invalid e-mail failure                Lorem ipsum  lipsum         foo123
foo123        Fail: invalid e-mail.
Name should not be empty              ${EMPTY}     eu@lipsum.com  foo123
foo123        Fail: name should not be empty.
Password should have 5 or more digits  Lorem ipsum  eu@lipsum.com  f123
f123          Fail: Password should have 5 or more digits.
Password should contain letters       Lorem ipsum  eu@lipsum.com  12345
12345         Fail: Password should contain letters.
Password should contain numbers       Lorem ipsum  eu@lipsum.com  foooo
foooo         Fail: Password should contain numbers.
Confirm password failure              Lorem ipsum  eu@lipsum.com  foo123
barr123       Fail: the confirm password doesn't match.

*** Keywords ***
Fill register form with data
  [Arguments]  ${name}  ${mail}  ${pass}
           ...  ${confirm_pass}  ${expected_message}

  Given I am on Bookstore Demo web site
    And I go to log in or sign in page
   When I fill "${name}" in name text field
    And I fill "${mail}" in e-mail text field
    And I fill "${pass}" in pass text field
    And I fill "${confirm_pass}" in confirm pass text field
    And I accept the terms of use
    And I submit register form
```

```
    Then I should see the "${expected_message}" message
     And I should see the register form
```

A variável `${EMPTY}` que está sendo utilizada na tabela de dados parece ter surgido sem ter sido declarada, mas na verdade existem algumas variáveis *built-in* que o Robot Framework disponibiliza automaticamente para facilitar a vida da pessoa programadora. A variável `${EMPTY}` representa uma variável vazia e sem ela não conseguiríamos representar um valor vazio em nossa tabela de dados. Existe também o recurso de lista vazia, através da variável `@{EMPTY}`, e de um dicionário vazio, através da variável `&{EMPTY}`. O mesmo ocorre com os valores booleanos, acessíveis através das variáveis `${TRUE}` e `${FALSE}`.

Também é possível acessar informações relativas ao contexto, tais como diretório corrente do arquivo da suíte, nome do caso de teste, *tags* e até mesmo o resultado da execução (neste caso, só é possível dentro do comando `Test Teardown`). Para conhecer todas as variáveis *built-in*, acesse https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#built-in-variables.

> Nomear as colunas como no exemplo anterior torna o teste mais fácil de entender. Isso é possível porque todas as informações em células ao lado da declaração de sessão são ignoradas pelo compilador.

Adicionamos ao final do script a palavra-chave `I should not see the Account page` para validar que realmente não foi possível realizar o cadastro e acessar a tela de acesso restrito do usuário. Sua implementação foi criada no arquivo `register.resource`, e o mapeamento do formulário está no arquivo de *page objects* `register_page_object.resource`:

```
I should see the register form
  Page Should Contain Element  ${REGISTRATION_FORM}
```

Assim como em suítes de testes *keyword-driven*, quando um caso de teste da tabela de dados falhar, os outros serão executados normalmente. Caso deseje alterar este comportamento, adicione a tag `robot:stop-on-failure` à suíte, de modo que, quando um caso de teste falhar, a execução seja interrompida.

Execute a suíte de testes e analise o relatório de execução. Agora ficou fácil criar novos cenários de teste, pois basta adicionar novas linhas na tabela de dados.

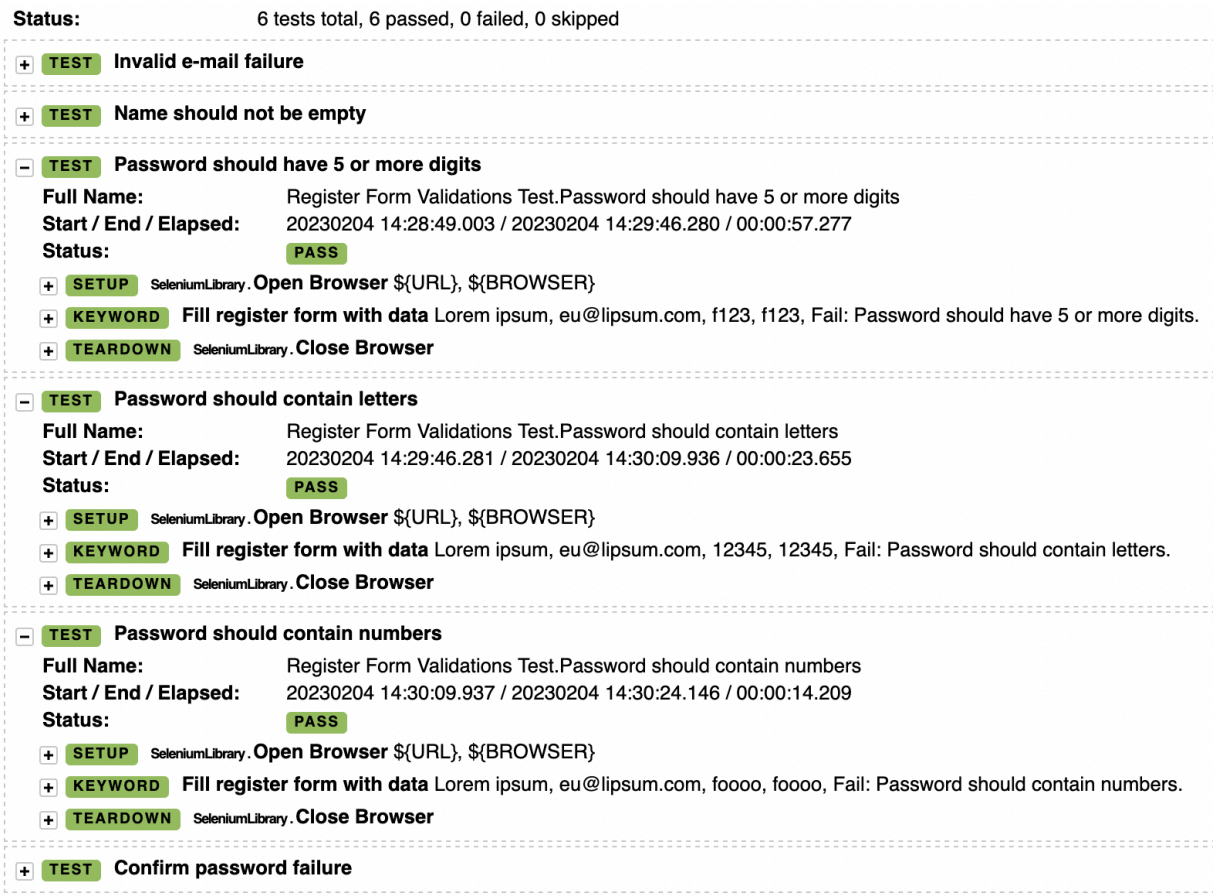


Figura 4.3: Logs da execução dos casos de teste dirigidos por dados.

4.3 Database: leitura em banco de dados

Suponhamos que trabalharemos em cenários de teste para testar a subscrição em lista de e-mails. Ao final do cadastro do e-mail, podemos visualizar e verificar uma mensagem de confirmação em tela, mas somente isso não garante que tudo ocorreu bem. Para estarmos seguros de que o contato de e-mail de fato entrou na lista, é necessário realizar uma leitura em banco de dados validando que ele está lá. Algo semelhante ocorre em nossos cenários de teste de cadastro. Ao final, estamos validando que é possível realizar a autenticação e visualizar as opções restritas do sistema, no entanto, podemos realizar uma consulta em base de dados para confirmar que o usuário está lá e que está com status ativo.

Em outra situação, esta mais recorrente, pode ser necessário buscar dados que satisfaçam determinadas pré-condições para a correta execução do fluxo de teste. Por exemplo, testes podem possuir como pré-condição de dados um produto sem estoque disponível, produtos cujo custo seja superior a determinado valor ou mesmo usuários inativos. Como proceder nessas situações?

A melhor estratégia é realizar leitura em banco de dados para buscar os dados necessários. O Robot Framework disponibiliza bibliotecas de acesso aos principais sistemas gerenciadores de bancos de dados, tanto relacionais quanto não relacionais. Em nosso projeto de testes da *Bookstore Demo*, trabalharemos com uma base de dados **MySQL**.

> Em alguns projetos, a política de segurança da empresa pode impedir que tenhamos acesso ao banco de dados do sistema. Quando não temos permissão de acesso ao banco de dados do ambiente de testes, há a possibilidade de clonar o banco de dados para uma base isolada somente para os testes automatizados realizarem consultas.

A biblioteca recomendada pelo Robot Framework para acesso em bancos de dados relacionais é a **DatabaseLibrary**, cuja documentação se encontra disponível em https://rpaframework.org/libraries/database/. Para acesso em bancos de dados não relacionais, as bibliotecas podem variar dependendo do tipo de banco de dados. A biblioteca dedicada ao MongoDB não é a mesma que funcionará com DynamoDB, por exemplo.

A instalação da DatabaseLibrary deve ser feita através do PIP, assim como todas as outras bibliotecas para Robot:

```
$ pip install robotframework-databaselibrary
```

Para trabalhar com bancos de dados, é necessário instalar um software intermediário que fará a comunicação entre o nosso script e o banco de dados em questão, que nada mais é do que um *driver* escrito em Python. Para trabalhar com o MySQL, por exemplo, precisaremos do `PyMySQL`.

```
$ pip install PyMySQL
```

Para configurar o acesso ao banco de dados, inseriremos algumas definições no arquivo de configuração do projeto, o `config.resource`:

```
*** Variables ***
${URL}        http://tst.bookstore.com/
${BROWSER}    firefox

${DB_NAME}    bookstoredb
${DB_USER}    database_user_name
${DB_PASS}    database_user_password
${DB_HOST}    localhost
${DB_PORT}    3306
```

A conexão com o banco de dados é criada através do comando `Connect to Database`. Recomendo criar um arquivo `database.resources`

dentro do diretório `resources/` para encapsular a conexão e as consultas ao banco de dados:

```
*** Settings ***
Documentation  Database connection and queries.
Library        DatabaseLibrary
Resource       config.resource

*** Keywords ***
Open database connection
  Connect To Database  pymysql  ${DB_NAME}  ${DB_USER}  ${DB_PASS}
${DB_HOST}  ${DB_PORT}
```

Atenção: não esqueça de fechar a conexão ao final do script, com o comando `Disconnect from database`. Utilize o `Test Teardown` para garantir que a conexão será devidamente encerrada ao final do teste.

```
[...]

Close database connection
  Disconnect from database
```

O principal comando para executar comandos SQL no banco de dados é o `Execute SQL String`, mas existem comandos específicos para cada necessidade. Caso a *query* seja muito grande, é uma boa prática separá-la em um arquivo `.sql`.

Validações em banco de dados

Vamos implementar uma melhoria em nossos cenários de teste de cadastro para validar que, ao final do cadastro, o usuário estará realmente ativo em banco de dados. Importaremos o arquivo `database.resource`, onde implementaremos as consultas, e adicionaremos ao final de cada cenário uma palavra-chave que realizará a consulta ao banco de dados.

Existem duas abordagens para abrir e fechar a conexão com a base de dados, uma reativa e outra propositiva. A forma reativa deixa a responsabilidade para o testador que estiver escrevendo a suíte de testes, que utilizaria os comandos `Test Setup` e `Test Teardown` para isso.

```
[...]

Resource        ../resources/database.resource

Test Setup      Open Browser  ${URL}  ${BROWSER}
                ...           AND  Open Database Connection

Test Teardown   Close Browser
                ...           AND  Close Database Connection

*** Test Cases ***
[...]

Scenario: Success sign in as Juridical Person
    Given I generated a fake e-mail
      And I am on Bookstore Demo web site
      And I go to log in or sign in page
     When I fill "Lorem ipsum LTDA" in name text field
      And I fill the fake e-mail in e-mail text field
      And I fill "12345" in pass text field
      And I fill "12345" in confirm pass text field
      And I select "Pessoa Jurídica" in business entity combo box
      And I accept the terms of use
      And I submit register form
     Then I should see the "Lorem ipsum LTDA, teu cadastro foi realizado
com sucesso!" message
      And I should see My Wish List
      And I should see My Orders
      And I should see the inserted user active in to the database
```

Contudo, para evitar problemas, prefiro a abordagem proativa: abrir e fechar a conexão a cada vez que necessitar executar uma *query*. Outra vantagem dessa abordagem é não abrir a conexão no início do

teste sem necessidade, pois se algo falhar antes da execução das *queries*, talvez a conexão tenha sido aberta em vão. Para isso, utilizaremos o recurso de `Teardown` dentro da *keyword*, que funciona de forma semelhante ao `Test Teardown` das suítes, sendo executado ao final da *keyword* independentemente de falhas.

A estrutura do banco de dados e a *query* em si não são relevantes para nosso estudo sobre Robot. Neste caso, a consulta que faremos simplesmente busca um registro na tabela de usuários do sistema que atenda a dois filtros: o e-mail cadastrado, que é único na base de dados, e a *flag* booleana `is_active`, que deve ser verdadeira. A implementação da palavra-chave que realiza a consulta e valida se o usuário está presente e ativo na base de dados fica assim:

```
[...]

I should see the inserted user active in to the database
  Open database connection
  Check If Exists In Database  SELECT email FROM users WHERE is_active = 1
AND email = '${FAKE_EMAIL}';
  [Teardown]                   Close database connection
```

Veja no relatório como a conexão foi encerrada imediatamente após a execução da palavra-chave que realiza a consulta.

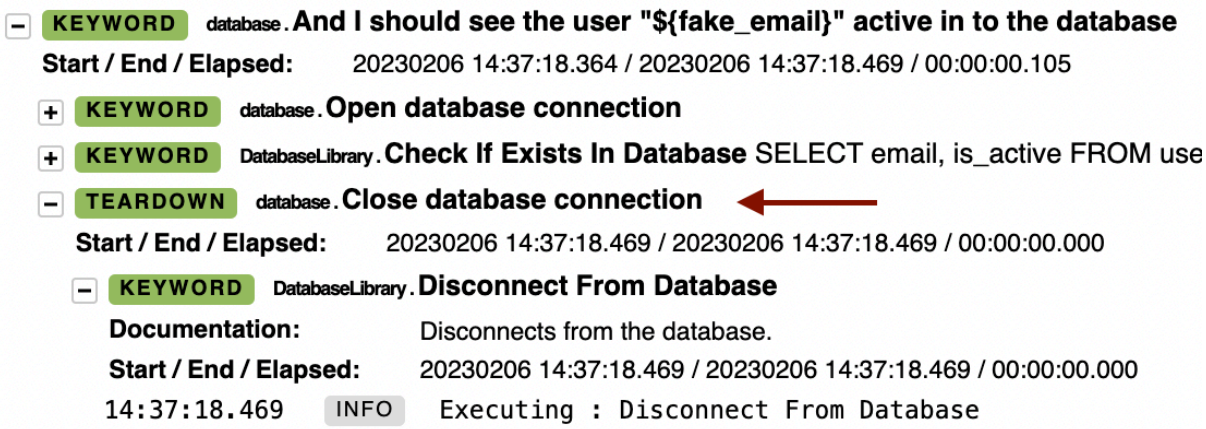


Figura 4.4: Encerramento da conexão com banco de dados no Teardown da palavra-chave.

Como esta consulta é bastante simples, podemos manter a consulta SQL onde está ou simplesmente configurá-la em uma variável com nome descritivo. Caso estejamos utilizando uma *query* maior e mais complexa, a recomendação é encapsular o SQL em arquivos separados, dentro do diretório `resources/queries/` .

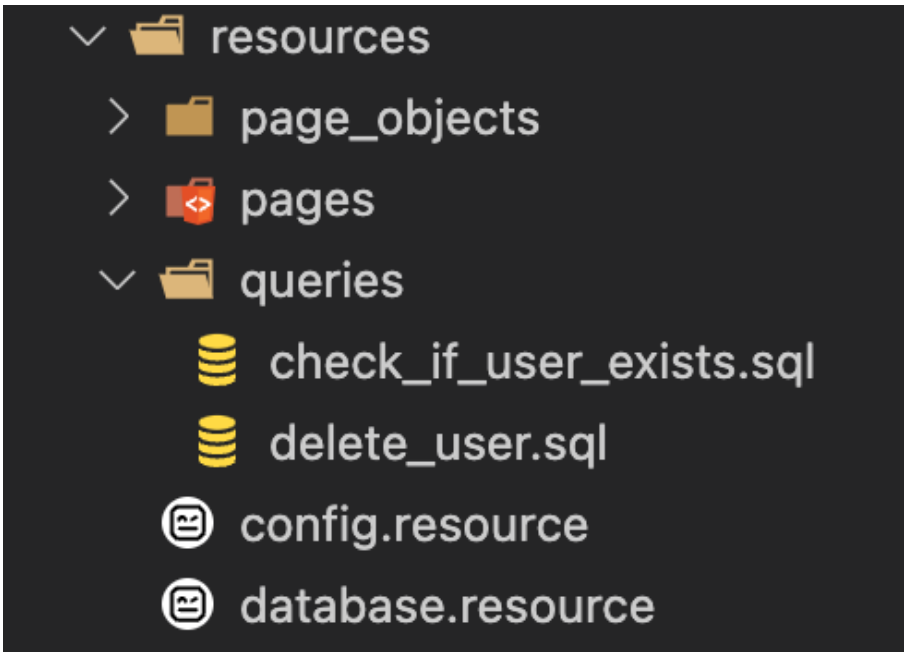


Figura 4.5: Organização das queries do projeto.

A interpolação de *strings* para passagem de parâmetros para a *query* pode ser realizada através do comando `Format String` da biblioteca String, que é padrão do Robot Framework e não necessita ser instalada. Dessa forma, inseriremos a consulta no arquivo SQL com as chaves `{}` marcando os valores que serão interpolados.

Um comando SQL para remoção de um usuário através do valor do campo e-mail, por exemplo, ficaria da seguinte forma:

```
DELETE FROM users WHERE email = '{user_mail}';
```

A interpolação pode ser posicional ou nomeada. No caso da abordagem posicional, os valores substituirão as lacunas na mesma

ordem em que foram passadas para a palavra-chave. Na abordagem nomeada, cada valor será inserido na lacuna correspondente ao nome definido na passagem do parâmetro. Em nosso caso, não fará diferença, visto que usaremos somente um argumento. No trecho de código a seguir, demonstro as duas formas de utilização equivalentes.

```
# Positional
{query} =  Format String   ${CURDIR}/queries/delete_user.sql
${user_email}

# Named
{query} =  Format String   ${CURDIR}/queries/delete_user.sql
user_mail=${user_email}
```

Expurgando registros em banco de dados

O acesso a banco de dados também pode ser útil para expurgar dados inseridos durante os testes com o objetivo de não deixar uma grande quantidade de dados inúteis após as execuções. No entanto, nem sempre teremos permissão para apagar registros na base de dados. Caso seja possível executar essa estratégia, o uso de um *dataprovider* para geração randômica de usuários se torna opcional.

Em cenários de teste como o de compra de determinado produto, por exemplo, após algumas execuções o estoque desse produto pode acabar. Chamamos isso de "queima de massa de dados". Nesse caso, é interessante executar uma *query* ao final do teste para repor o estoque do produto, ou seja, desfazer o que foi feito pelo teste.

No caso dos nossos cenários de teste de cadastro, basta removermos o usuário da tabela de usuários. O sistema da Bookstore Demo é bastante simples, mas em sistemas mais complexos é provável que seja necessário realizar a remoção de dados em mais de uma tabela. Crie a palavra-chave `I should delete the user "${user_email}" from database` e adicione-a ao final dos

cenários de teste de cadastro. Utilizaremos o comando `Execute SQL String` da DatabaseLibrary para executar o `DELETE`, que removerá o usuário da base de dados.

```
I should delete the inserted user from database
    Open database connection
    Execute SQL String   DELETE FROM users WHERE email = '${FAKE_EMAIL}';
    [Teardown]           Close database connection
```

Buscando dados qualificados

Construiremos um cenário de teste para validar que não é possível comprar um produto que esteja fora de estoque e o sistema emitirá a mensagem apropriada. Para construir este script, precisaremos buscar em banco de dados um produto que satisfaça a pré-condição, ou seja, que esteja sem estoque disponível. O cenário deverá atender à seguinte história de usuário:

```
Como cliente da loja virtual da livraria Bookstore Demo, desejo saber
quando um produto está fora de estoque.

  Dado um determinado produto fora de estoque
     E que estou na loja virtual da Bookstore Demo
Quando busco por este determinado produto através do seu ISBN
 Então devo visualizar a mensagem "Produto indisponível"
     E devo visualizar o botão de Adicionar ao Carrinho desabilitado
     E devo visualizar o botão de Comprar Agora desabilitado
```

Iniciaremos criando um novo arquivo dentro do diretório `tests/`, nomeado `stock_test.robot`, onde construiremos a suíte de testes da funcionalidade de estoque da loja. Assim como nas suítes anteriores, devemos importar a biblioteca do Selenium e o arquivo de configurações do projeto. Como neste caso utilizaremos recursos de acesso a banco de dados, também devemos importar o arquivo `database.resource`. As definições de `Test Setup` e `Test Teardown` serão aquelas com que já estamos acostumados, ou seja, abrir o navegador e fechar navegador, respectivamente. Na seção de casos de teste, escreveremos o passo a passo do cenário.

```
*** Settings ***
Documentation   Como cliente da loja virtual da livraria Bookstore Demo,
          ...   desejo saber quando um produto está fora de estoque.

Resource        ../resources/config.resource
Resource        ../resources/database.resource
Resource        ../resources/pages/home.resource
Resource        ../resources/pages/product.resource

Test Setup      Open Browser  ${URL}  ${BROWSER}
Test Teardown   Close Browser

*** Test Cases ***
Scenario: Product out of stock
  Given I have a book out of stock
    And I am on Bookstore Demo web site
   When I search for this book by ISBN
   Then I should see the "Produto indisponível" message
    And I should see the Add to Cart button disabled
    And I should see the Buy Now button disabled
```

O primeiro passo do script é acessar a base de dados para buscar um produto que esteja fora de estoque. Implementaremos uma *keyword* no arquivo `database.resource` que realizará uma consulta no banco de dados através do comando `Query` , da DatabaseLibrary. Essa consulta deve retornar uma lista de produtos fora de estoque, da qual utilizaremos o primeiro item. Caso nenhum seja encontrado e a lista esteja vazia, o teste falhará com a mensagem *None valid product found in database*, pois, sem massa de dados disponível, não é possível seguir.

Seria possível adotar a abordagem de o próprio script inserir um registro em banco de dados que atenda à necessidade, mas não faremos dessa forma neste exemplo. Em vez de retornar o valor, o dado será gravado em uma variável de teste para utilização no caso de teste, por conta da estratégia detalhada anteriormente neste capítulo, na seção *Dataprovider: geração dinâmica de massa de dados*.

```
[...]

I have a book out of stock
  Open database connection
  @{resultset} =       Query                SELECT isbn FROM books WHERE
stock = 0;
  Should Not Be Empty  item=@{resultset}    msg=None valid product found
in database.
  ${book_item} =       Set Variable         @{resultset}[0]
  Set Test Variable    ${OUT_OF_STOCK_BOOK}  ${book_item}
  [Teardown]           Close database connection
```

A busca do produto foi implementada no arquivo `resources/page/home.resource`. O ISBN será inserido na barra de pesquisa de produtos da página principal do site e, em seguida, a tecla ENTER é pressionada. Como o ISBN é único, em vez de visualizar a lista de resultados da busca, o site nos redirecionará diretamente para a tela do produto.

```
*** Settings ***
Library   SeleniumLibrary
Resource  ../page_objects/home_page_object.resource

*** Keywords ***
I am on ${expected} web site
  Title Should Be  ${expected}

I search for a book by ${search_text}
  Input Text  ${BOOKS_SEARCH_BAR}  ${search_text}
  Press Keys  ${BOOKS_SEARCH_BAR}  RETURN

I search for this book by ISBN
  Input Text  ${BOOKS_SEARCH_BAR}  ${OUT_OF_STOCK_BOOK}
  Press Keys  ${BOOKS_SEARCH_BAR}  RETURN
```

O mapeamento da barra de pesquisas está na camada de *page objects*, no arquivo `resources/page_objects/home_page_object.resource`.

```
*** Variables ***
${BOOKS_SEARCH_BAR}  name: search-bar
```

Por fim, os pontos de verificação do teste serão implementados no arquivo `resources/pages/product.resource`. São três: aguardar que a mensagem seja exibida e verificar que os botões de "Adicionar ao Carrinho" e "Comprar Agora" estejam desabilitados, como na tela a seguir:

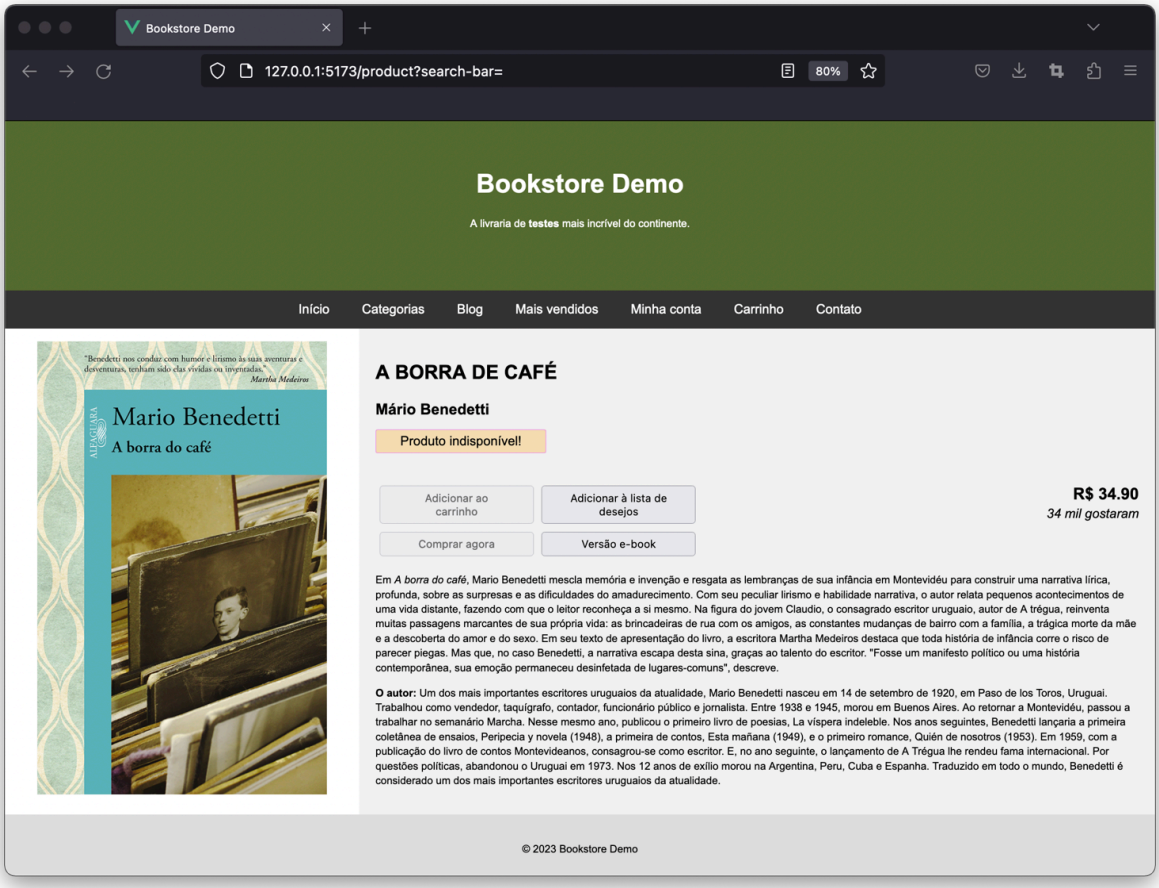


Figura 4.6: Tela de produto fora de estoque.

Faremos isso através dos comandos `Wait Until Page Contains` e `Element Should Be Disabled` do Selenium.

```
*** Settings ***
Library   SeleniumLibrary
Resource  ../page_objects/product_page_object.resource

*** Keywords ***
```

```
I should see the "${message}" message
  Wait Until Page Contains   ${message}

I should see the Add to Cart button disabled
  Element Should Be Disabled  ${ADD_TO_CART_BUTTON}

I should see the Buy Now button disabled
  Element Should Be Disabled   ${BUY_NOW_BUTTON}
```

Os mapeamentos dos objetos, logicamente, estarão em `resources/page_objects/product_page_object.resource` :

```
*** Variables ***
${ADD_TO_CART_BUTTON}  name: add-to-cart
${BUY_NOW_BUTTON}      name: buy-now
```

Execute o teste e analise o fluxo dos dados através do relatório de execução.

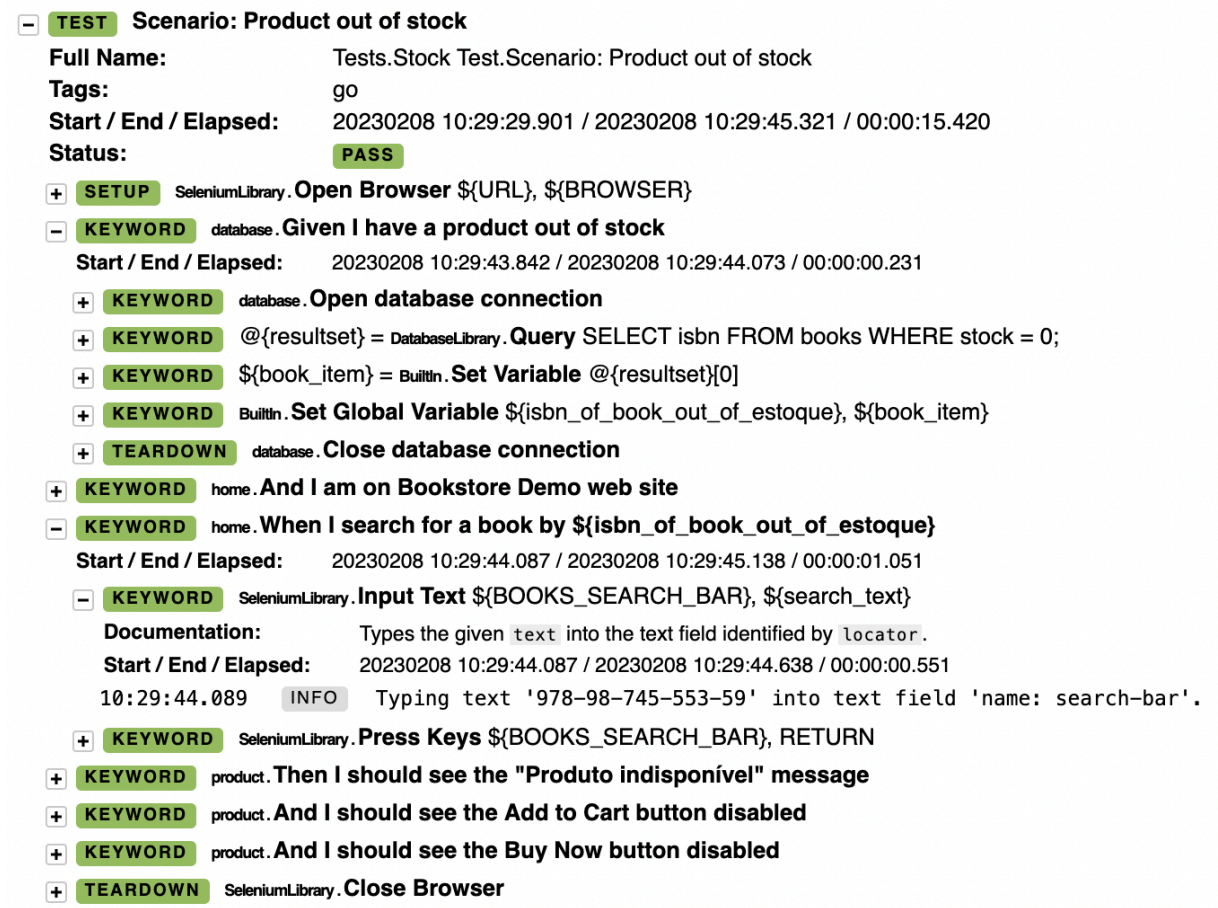


Figura 4.7: Logs da execução do cenário.

Analise atentamente o contexto de cada projeto, suas limitações e possibilidades, e defina uma maneira de lidar com a massa de dados.

A estratégia adotada para gerenciar a massa de dados é, certamente, uma das decisões mais importantes para o sucesso do projeto de automação. A implementação dessas estratégias pode ser mais complicada do que a construção do fluxo de teste em si e, por esse motivo, é comum que seja postergada ou até mesmo negligenciada. Uma abordagem cuidadosa é fundamental para que a bateria de testes seja resiliente e confiável. Testes que dependem de intervenção manual para preparar massa de dados fatalmente serão abandonados por não serem eficientes.

Capítulo 5
Testes de API

API significa *Application Programming Interface* (Interface de Programação de Aplicação). Segundo o *Hub de conceitos* da Amazon Web Services, as APIs são mecanismos que permitem que dois componentes de software se comuniquem usando um conjunto de definições e protocolos.

Existem várias formas de implementar uma API. É muito comum encontrarmos sistemas orientados à arquitetura de serviços (SOA, do inglês *Service-Oriented Architecture*), que podem ser implementados através de SOAP, REST, RPC etc. Também é recorrente encontrarmos sistemas que utilizam filas de mensagens para se comunicar, por exemplo, com Apache Kafka. Para cada arquitetura, teremos uma estratégia de testes diferente e uma biblioteca específica para utilizar.

Para testes em SOAP ou REST, utilizaremos a RequestsLibrary (https://marketsquare.github.io/robotframework-requests/doc/RequestsLibrary.html).

Para testes com Kafka, utilizaremos KafkaLibrary (https://rawgit.com/s4int/robotframework-KafkaLibrary/master/doc/KafkaLibrary.html).

Como uma ferramenta auxiliar, também veremos como acessar mensagens de e-mail através da biblioteca ImapLibrary. É comum que sistemas enviem *tokens* ou códigos de confirmação por correio eletrônico para o usuário, e não é eficiente nem seguro acessar uma caixa de e-mails via interface para validar o correto funcionamento desse mecanismo. Essa tarefa pode ser realizada através do protocolo IMAP (*Internet Message Access Protocol*), que é um

protocolo de gerenciamento de correio eletrônico. A biblioteca utiliza esse protocolo para que o Robot Framework possa acessar uma caixa de e-mails, monitorar a chegada de novas mensagens e analisar seu conteúdo, sem a necessidade de interface gráfica. Para entender mais sobre como a biblioteca ImapLibrary funciona, leia a documentação oficial, que está em https://lasselindqvist.github.io/robotframework-imaplibrary2/doc/ImapLibrary2.html.

É importante cobrir as APIs com testes de segurança e de desempenho; no entanto, o Robot Framework não é a ferramenta adequada para esses tipos de teste. Com o Robot, focaremos em garantir a cobertura de *testes funcionais* e *testes de contrato*.

Mas por que investir em testes de API? Segundo Martin Fowler (2012b), os testes executados de ponta a ponta por meio da interface do usuário são mais frágeis, caros para escrever e demorados para executar, em comparação aos testes de API. Ainda segundo Fowler, o teste de serviço fornece muitas das vantagens do teste de ponta a ponta, mas evita as complexidades de lidar com a interface de usuário.

Como falamos no capítulo 1, na seção *Quando e o que automatizar* sobre a pirâmide de testes, os testes de API são mais simples e rápidos de construir, manter e executar, além de ser um "atalho" para identificar *bugs* que se replicariam na camada de interface em todos os sistemas que consomem a mesma API. Visualizando o modelo de arquitetura de sistemas orientados a serviços na imagem a seguir, fica mais nítida a vantagem de investir em testes em nível de integração.

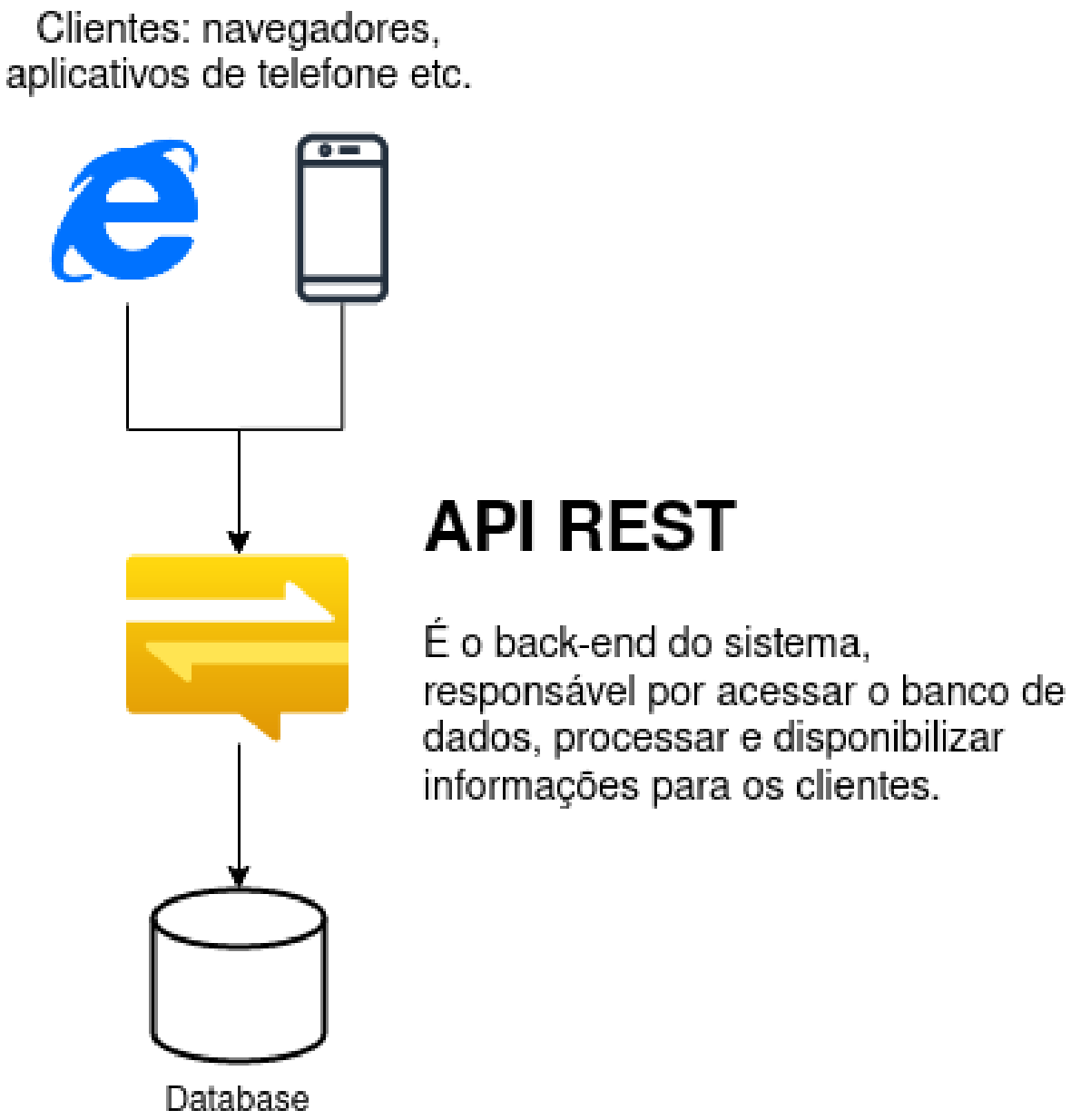


Figura 5.1: Arquitetura da loja virtual da Bookstore Demo.

Em nosso caso, como estratégia de testes para o sistema da livraria virtual, exercitaremos testes sob uma API de serviços REST, afinal, essas são as APIs mais populares e flexíveis encontradas na Web atualmente, conforme o *Hub de conceitos* da Amazon Web Services. A arquitetura desse sistema é bastante simples: a API de serviços REST realiza a comunicação entre o servidor de banco de dados que

roda no *back-end* e os sistemas de *front-end*, como um aplicativo mobile e o site web que já conhecemos e testamos com o Selenium.

Para compreender como construir testes automatizados para API de serviços, é fundamental conhecer o protocolo HTTP, que é o protocolo que define as regras de comunicação entre cliente e servidor na internet. Conhecer os métodos de requisição e as estruturas da mensagem são essenciais. Se você quiser se aprofundar no funcionamento do modelo arquitetural de cliente-servidor e do protocolo HTTP, recomendo o livro *HTTP: The Definitive Guide*, de David Gourley e Brian Totty (O'Reilly, 2002), e o curso *HTTP: Entendendo a web por baixo dos panos*, da Alura (www.alura.com.br). A especificação oficial do protocolo fica no site do W3C, em https://www.w3.org/Protocols/.

Neste modelo, a comunicação ocorre através de troca de mensagens. O que nosso teste deve fazer é enviar uma mensagem ou uma série de mensagens ao servidor e validar a sua resposta. Para realizar uma requisição, deve-se utilizar um método, que pode ser GET, POST, PUT, PATCH, dentre outros. Cada um deles possui um objetivo.

É muito bom quando o time de desenvolvimento possui uma documentação completa dos serviços, como um Swagger ou uma coleção do Postman, mas nem sempre poderemos contar com isso. O time de desenvolvimento da Bookstore Demo, por sua vez, disponibilizou uma coleção de requisições do Postman cobrindo toda API disponível e que será suficiente para o nosso trabalho.

A coleção do Postman para a API da Bookstore Demo está disponível no diretório `tests/` do repositório https://github.com/thvieira/bookstore-api-demo.

A seguir, os serviços relacionados com a funcionalidade de cadastro de livros, que será a primeira na qual trabalharemos:

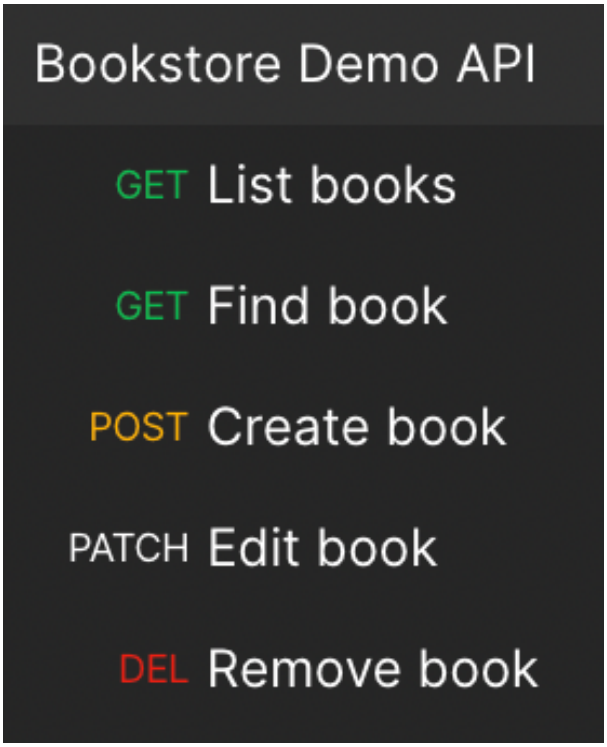


Figura 5.2: API de serviços da Bookstore Demo.

Agora que conhecemos melhor a arquitetura do sistema que estamos testando, seguiremos para a implementação. As requisições HTTP serão realizadas através da biblioteca RequestsLibrary, cuja documentação está em https://marketsquare.github.io/robotframework-requests/doc/RequestsLibrary.html.

5.1 Testes de contrato

Este é um tipo de teste de integração normalmente utilizado para garantir que serviços externos ao time não provocarão problemas na aplicação por modificarem seu contrato. Segundo Martin Fowler, uma falha em um teste de contrato não deve necessariamente interromper a implantação de versão da mesma forma que uma falha de teste normal faria, mas, sim, gerar uma nova tarefa para o

time realizar os ajustes necessários para manter o código consistente novamente.

A verificação da correta utilização de códigos de status do HTTP diante de cada caso de teste, seja de exceção ou não, é responsabilidade dos testes de contrato. É importante o testador conhecer os códigos HTTP e o objetivo de cada um. Sugiro a leitura da documentação disponível no site da Mozilla, em https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. Contudo, além do código de status de resposta, segundo Wojciech Bulaty no artigo *Testing microservices* (2019), os testes de contrato de serviços devem validar tanto os formatos de mensagens entre os componentes (sintaxe) quanto as expectativas comportamentais dos componentes (semântica).

Construiremos agora os testes de contrato para o serviço `/books`. Podemos considerar um levantamento simplificado de casos de teste, como a tabela a seguir. Não considere que, com este exemplo, cobriremos todas as possibilidades e cenários de teste, pois o objetivo aqui é apenas demonstrar o uso dos recursos da ferramenta.

ID do item	Serviço	Caso de teste	Status esperado	Conteúdo esperado
B001	GET /books	Listar todos os livros	200 OK	Lista com todos os livros da base de dados.
B002	GET /books?search	Listar livros com filtro (ISBN, título ou autor)	200 OK	Lista com livros que atendam à busca.

ID do item	Serviço	Caso de teste	Status esperado	Conteúdo esperado
B003	GET /books?search	Listar livros com filtro (ISBN, título ou autor)	204 No content	Lista vazia.
B004	GET /books?id	Buscar detalhes de um livro por id	200 OK	Dados do livro.
B005	GET /books?id	Buscar detalhes de um livro inexistente	404 Not found	Mensagem "Item not found".
B006	POST /books	Cadastrar livro	201 Created	ID do livro cadastrado.
B007	POST /books	Cadastrar livro com ISBN repetido	409 Conflict	Mensagem "ISBN already in the data base".
B008	POST /books	Cadastrar livro com dados inválidos ou omitidos	400 Bad request	Mensagem "Invalid data! You must fill ISBN, author, title, publisher and price".
B009	POST /books	Cadastrar livro sem estar autenticado	401 Unauthorized	Mensagem "You have to be

ID do item	Serviço	Caso de teste	Status esperado	Conteúdo esperado
				authenticated to do it".
B010	POST /books	Cadastrar livro com autenticado como cliente e não como admin	403 Forbidden	Mensagem "You don't have permission to do it".
B011	PATCH /books	Editar dados de um livro	200 OK	Dados do livro editado.
B012	PATCH /books	Editar livro com dados inválidos	400 Bad request	Mensagem: "Inconsistent data".
B013	PATCH /books	Editar dados de um livro inexistente	404 Not found	Mensagem "Item not found".
B014	DELETE /books?id	Remover um livro	204 No content	Livro removido da base de dados.
B015	DELETE /books?id	Remover uma lista de livros	204 No content	Livros removidos da base de dados.

Normalmente, todos esses testes poderiam ser implementados em um único arquivo `books_api_test.robot`, dedicado aos testes do serviço *books*. Contudo, para melhor organização dos exemplos, separaremos os casos de teste em quatro arquivos:

- `books_list_api_test.robot`
- `books_creation_api_test.robot`
- `books_edition_api_test.robot`
- `books_deletion_api_test.robot`

Esses arquivos ficarão no diretório `/tests/api/contract/`. Perceba que adoto o sufixo `_api_test` para a nomenclatura das suítes de teste de API.

O primeiro passo é adicionar a URL do servidor da API no arquivo de configurações do projeto, o `config.resource`.

```
*** Settings ***
Documentation  Project global definitions.

*** Variables ***
${URL}            http://tst.bookstore-demo.com/
${BROWSER}        chrome
${API_HOST}       http://localhost:3000/v1/

[...]
```

O arquivo de configurações deve ser importado na seção *Settings* dos arquivos das suítes de teste, da mesma forma que fizemos nos testes web. Também utilizaremos o `Documentation` para documentar o arquivo, o comando `Test tags` para definir algumas *tags* para a suíte e importar a biblioteca `RequestsLibrary`.

GET: Acessando informações

O método `GET` serve para requisitar a representação atual de determinado item ou conjunto de itens, sem alterar o estado atual desses recursos. A implementação do caso de teste B001, *List all books*, utiliza o método `GET`, pois disponibiliza a lista de livros disponíveis na base de dados, na verdade um objeto JSON representando o conjunto atual de itens.

A implementação dessa requisição se dá através da `RequestsLibrary`, que é a biblioteca para construção de requisições HTTP no Robot Framework. A palavra-chave é intitulada "GET", simplesmente, e deve obrigatoriamente receber como parâmetro a URL do serviço, neste caso, `${API_HOST}/books`. Existem alguns parâmetros opcionais que veremos mais adiante, como o `expected_status`, por exemplo, o qual serve para validar o código de status esperado da resposta.

A palavra-chave retorna os dados da resposta, que devem ser atribuídos a alguma variável. Em seguida, verificamos se o conteúdo da resposta possui uma lista nomeada `books` utilizando recursos da biblioteca `Collections`, que é padrão do Robot e não precisa ser instalada. O comando `Evaluate` executa uma expressão Python, neste caso, para validar que um objeto é do tipo lista, através da função nativa `isinstance()`.

```
*** Settings ***
Documentation  Bookstore Demo
          ...  Books List API Contract Testing
Library        Collections
Library        RequestsLibrary
Resource       ../../../resources/config.resource
Test Tags      api  books  list

*** Test Cases ***
Scenario: List all books
  [Tags]   B001
```

```
  # Get books list
  ${response} =  GET  ${API_HOST}/books  expected_status=200

  # Check if the response has a book list
  Dictionary Should Contain Key  ${response.json()}  books
  ${is_list} =  Evaluate     isinstance(${response.json()}[books], list)
  Should Be True  ${is_list}

[...]
```

Agora devemos refatorar essa implementação. O código está um pouco confuso e a própria existência dos comentários demonstram que esse script pode ser dividido e reorganizado. Neste caso também podemos utilizar o Gherkin, criando palavras-chave novas para agrupar conjuntos de passos e tornar o cenário de teste mais legível e coeso. Neste processo, separaremos a validação do código de status da chamada da requisição, criando um passo específico para esta validação, para que fique mais claro o que está acontecendo.

```
*** Test Cases ***
Scenario: List all books
  [Tags]  B001
  When I get books list
  Then I should see a books list
   And I should see response status code 200

[...]

*** Keywords ***
I get books list
  ${response} =  GET  ${API_HOST}/books
  Set Test Variable  ${RESPONSE}  ${response}

I should see a books list
  Dictionary Should Contain Key  ${RESPONSE.json()}  books
  ${is_list} =  Evaluate  isinstance(${RESPONSE.json()}[books], list)
  Should Be True  ${is_list}
```

```
I should see response status code ${expected_status}
  Status Should Be  ${expected_status}  ${RESPONSE}
```

Para construir as requisições dos serviços de busca com filtros ou de busca por detalhes de um item específico, precisaremos utilizar parâmetros em nossas requisições. Segundo especificação do HTTP, existem diferentes formas de enviarmos dados como parâmetro para as requisições GET, como *query parameters*, *path parameters* e *header parameters*.

Query parameters

Para construir uma requisição com filtro na lista de livros, implementando uma busca por título, nome de autor ou ISBN, por exemplo, devemos utilizar o *query parameter* `search` do serviço `/books`. Esse tipo de estrutura é comum em diferentes contextos onde existe possibilidade de filtros de busca. O retorno da requisição `GET /books?search=978-1-56619-909-4` resultaria em algo como:


```
{
    "books": [
        {
            "id": 9842,
            "isbn": "978-1-56619-909-4",
            "title": "El canto del viento",
            "author": "Atahualpa Yupanqui",
            "publisher": "Editora Coragem",
            "price": "50.00",
            "stock": 0
        }
    ]
}
```


Query parameters, literalmente "parâmetros de consulta", são o tipo de parâmetro mais comum. Eles são inseridos ao final da URL, antecedidos por um sinal de interrogação `?`. Nome e valor são passados no formato `name=value` e separados pelo caractere `&`.

Dependendo do contrato de cada serviço, podem ser obrigatórios ou não.

Para demonstrar como utilizar os *query parameter* no Robot Framework, construiremos o cenário de teste B002, *List books with filter by ISBN*. Inicialmente, geraremos um livro com dados randômicos e o inseriremos no banco de dados. Em seguida, realizaremos a busca através do serviço `/books`, passando o ISBN como parâmetro de consulta. O código de status esperado é *200 Ok*. O mesmo livro deve ser retornado, e, como o ISBN é único, a lista deve ter tamanho 1. Ao final, removemos os dados inseridos do banco de dados.

```
Scenario: List books with filter by ISBN
  [Tags]  B002  ISBN
  Given I get a fake book
    And I insert the fake book into database
   When I search for the fake book by ISBN
   Then I should see response status code 200
    And I should see the fake book in books list response
    And I should see 1 book in books list response
   [Teardown]  I should remove the fake book from database
```

A passagem dos parâmetros é realizada na palavra-chave `I search for the fake book by ISBN`. Os métodos de requisições da biblioteca RequestsLibrary aceitam um parâmetro nomeado `params`, que recebe os *query parameters*, os quais podemos inserir diretamente na chamada da palavra-chave ou definir separadamente em um dicionário. Eu recomendo a segunda opção, para melhor organização do código e do projeto. Pode não parecer vantajoso quando temos somente um parâmetro, mas ainda assim existe ganho nesta abordagem. Como a sintaxe de definição de nome e valor dos parâmetros é igual à sintaxe de definição dos parâmetros das palavras-chave do Robot, podemos causar alguma confusão ao construir a passagem de parâmetros desta forma:

`params=search=${value}`.

```
[...]

I search for the fake book by ISBN
  &{query_params} =  Create Dictionary       search=${FAKE_BOOK.isbn}
  ${response} =      GET  ${API_HOST}/books  params=&{query_params}
  Set Test Variable  ${RESPONSE}             ${response}
```

O relatório do Robot mostra em detalhes tudo o que aconteceu:

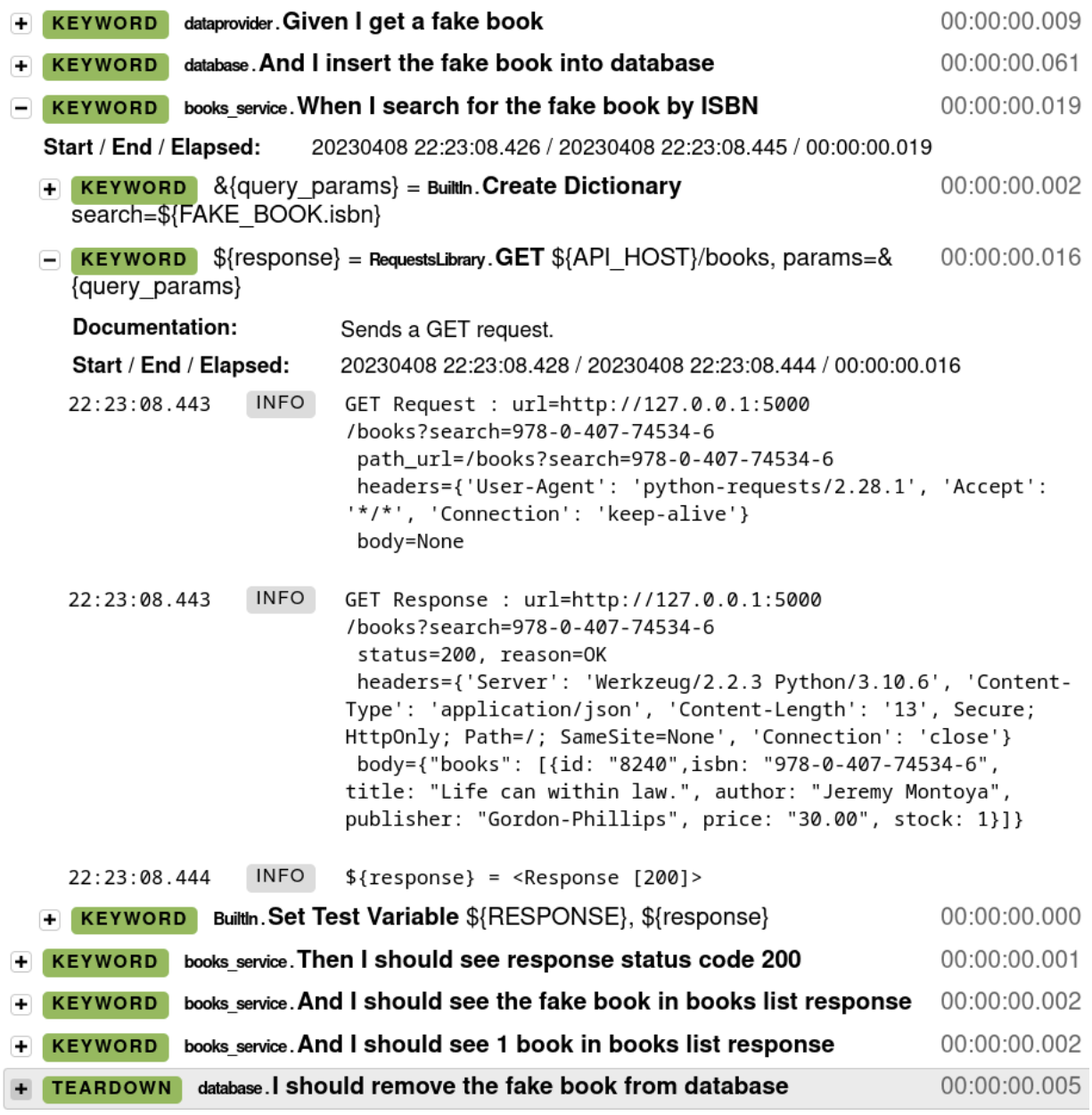


Figura 5.3: List books with filter by ISBN: query parameter.

Path parameters

Path parameters são partes variáveis da própria URL do serviço. Normalmente são utilizados para acessar um recurso específico dentro de uma coleção de recursos, exatamente como é o caso do cenário de teste *Find book by ID*. Geralmente, a representação da parte variável da URL, ou seja, o *path parameter*, é feita entre chaves. No caso do serviço que testaremos agora, fica `GET /books/{id}`.

Assim como no cenário anterior, inseriremos um livro com dados randômicos em banco de dados para executar o teste e o removeremos ao final. Após a requisição `GET /books/{id}`, validaremos o código de status e o conteúdo da resposta, verificando se os dados estão corretos.

```
Scenario: Find book by ID
  [Tags]   B004
  Given I get a fake book
    And I insert the fake book into database
   When I get the fake book by ID
   Then I should see response status code 200
    And I should see the fake book informations
  [Teardown]  I should remove the fake book from database
```

Na implementação da palavra-chave `I get the fake book by ID`, simplesmente inserimos o ID do livro ao final da URL do serviço.

```
[...]


I get the fake book by ID
  ${response} =      GET  ${API_HOST}/books/${FAKE_BOOK.id}
  Set Test Variable  ${RESPONSE}  ${response}
```

Vejamos no relatório como ocorreu a requisição:

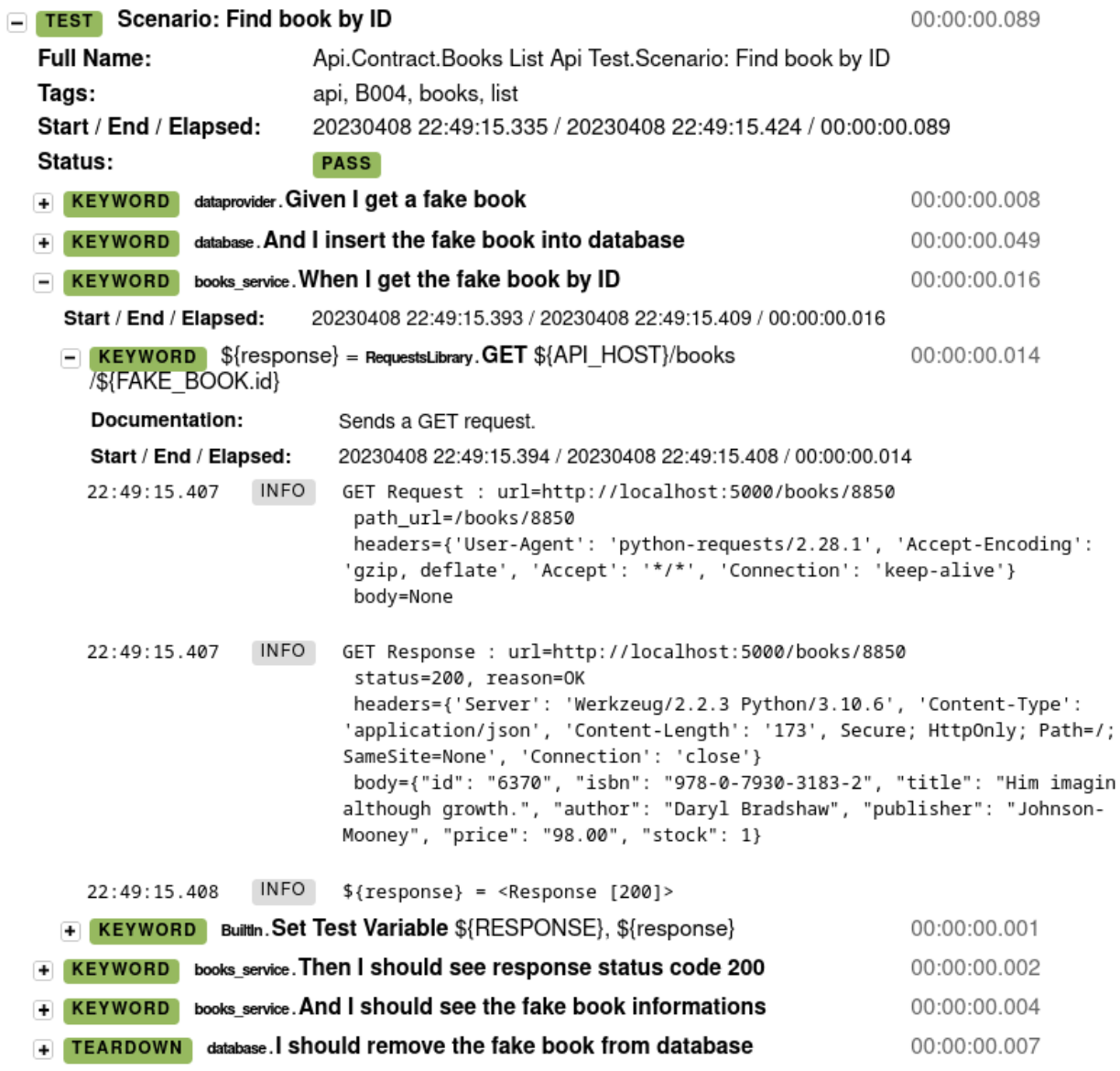


Figura 5.4: Find book by ID: path parameter.

POST: Criando novos itens

Para escrever os cenários de teste de cadastro de livros no sistema, passaremos para o arquivo `books_creation_api_test.robot`. Para criar novos recursos em um sistema através de uma API REST, utiliza-se o método `POST`.

O funcionamento desse tipo de requisição no Robot é parecido com o `GET`. A principal diferença consiste na existência de um conteúdo a

ser enviado no campo `body` da mensagem HTTP, contendo os dados do livro que queremos inserir na base de dados da loja. Estes dados também são chamados de *payload*.

No caso da nossa API REST, estes dados possuem formato JSON, o formato mais comum nesse tipo de API, mas isso não é uma regra. Em outras APIs, o formato dos dados pode ser XML, texto puro ou até mesmo outros formatos menos usuais. O código de status esperado neste caso será *201 Created*, que indica que o novo recurso foi criado corretamente. Normalmente, a resposta da requisição nos informa o identificador do recurso que foi criado.

Para melhor organização do código, inserimos o *payload* das nossas requisições em arquivos JSON separados, em um diretório auxiliar, `resources/payloads/` . Para acessar o conteúdo dos arquivos, basta utilizar o comando `Load Json From File` da biblioteca JSONlibrary. Para alterar informações, se necessário, podemos utilizar o comando `Set to dictionary` da biblioteca Collections. Para passar este conteúdo do arquivo JSON para a requisição, devemos utilizar o parâmetro `json` da palavra-chave `POST` .

Um código-fonte inicial com as informações que temos até agora ficaria assim:

```
*** Test Cases ***
Scenario: Create book
  [Tags]               B006
  ${body} =            Load  Json From File
resources/payloads/create_book.json
  ${respose} =         POST  ${BOOKSTORE_API}/books  json=${body}
expected_status=201
  ${inserted_id} =     ${response.json()}[id]
  Should Not Be Empty  ${inserted_id}
  Check If Exists In Database  SELECT id FROM books WHERE id =
'${inserted_id}';
```

Todos os passos do caso de teste parecem estar contemplados nestas linhas. Execute-o utilizando a *tag* B006 para vê-lo

funcionando.

Cabeçalho de requisição

Infelizmente, ao executar o teste que acabamos de escrever obteremos uma falha devido ao código de *status* da resposta ser *401 Unauthorized* em vez de *201 Created*, como era esperado. Isso ocorre porque este serviço é protegido, sendo necessário fazer a autenticação no sistema.

Existe um serviço para autenticação no sistema da Bookstore, o `POST /auth`. Podemos utilizá-lo para realizar a autenticação antes das requisições que serão testadas, como pré-condição do teste. Os dados de autenticação na API podem ficar no arquivo `config.resources`, mas, como são dados sensíveis, o ideal seria que estivessem armazenados em algum serviço de *key-store*.

O serviço de autenticação também é uma requisição POST, e devemos enviar um nome de usuário e uma senha no corpo da requisição. Este *payload* tem o seguinte formato:

```
{
  "username": "bookstore_admin",
  "password": "admin123"
}
```

A resposta deverá vir com uma mensagem de sucesso e o token que será utilizado no cabeçalho das nossas requisições para autenticação. Normalmente, as sessões possuem limite de tempo de alguns minutos até expirar, forçando a necessidade de emissão de um novo token, mas esse detalhe não interferirá nos testes.

```
{
    "message": "Successful log in as bookstore_admin.",
    "token": "2137ab8976e4472abc07a312a49e81f3"
}
```

O token de autenticação deve ser enviado no cabeçalho da requisição HTTP. Para preparar os dados que serão enviados no

cabeçalho, podemos utilizar um dicionário do Robot, de forma semelhante ao que fizemos para trabalhar com *query parameters*. O nome do cabeçalho que deve receber o token de autenticação, no caso dos serviços desta API especificamente, é `x-session`. Através da palavra-chave `Create Dictionary` da biblioteca `Collections`, criamos um dicionário com o token recém-gerado. Criaremos então o dicionário *headers* com o token e também com a informação de *Content type*, que serve para informar qual é o formato dos dados da requisição, e enviaremos este dicionário através do parâmetro `headers` para a requisição. As informações contidas no cabeçalho também são chamadas de *header parameters*. Sempre que for necessário enviar alguma informação no cabeçalho das requisições HTTP, devemos proceder dessa forma.

```
*** Test Cases ***
Scenario: Create book
  [Tags]  B006

  # Autenticação
  &{auth_payload} =   Create Dictionary  username=${API_USER}
password=${API_PASS}
  ${auth_response} =  POST  ${API_HOST}/auth  json=&{auth_payload}
expected_status=200
  ${token} =          Set Variable  ${auth_response.json()}[token]

  # Criação do novo livro
  ${body} =           Load Json From File
resources/payloads/create_book.json
  &{header} =         Create Dictionary  x-session=${token}  Content-
Type=application/json
  ${response} =       POST  ${API_HOST}/books  json=${body}  headers=&
{header}  expected_status=201
  ${inserted_id} =    Set Variable  ${response.json()}[id]

  # Validação da entrada do novo livro na base de dados
  Should Not Be Empty  ${inserted_id}
  Open database connection
```

```
  Check If Exists In Database  SELECT isbn FROM books WHERE id =
'${inserted_id}' and isbn = '${isbn}';
```

Agora, sim, nosso teste funcionou! A autenticação foi realizada corretamente, inserimos os dados do novo livro no banco de dados e obtivemos o ID deste item no corpo de resposta da requisição, que foi realizada com código de status *201 Created*. Todavia, experimente executar o mesmo script uma vez mais. Agora nosso teste falhará, desta vez porque a resposta da requisição apresenta código de status *409 Conflict*. Por que funcionou uma vez e agora não funciona mais?

O código HTTP 409 indica que houve algum conflito. Neste caso, trata-se do ISBN repetido. Aliás, deve haver um caso de teste validando que esta regra de negócio está sendo corretamente implementada pelo microsserviço. Se não modificarmos o número de ISBN cada vez que executarmos esse script, haverá essa mesma falha. Esse problema pode ser resolvido de duas formas: utilizando a biblioteca Faker para gerar números de ISBN randômicos ou removendo os dados criados do banco de dados após cada execução do teste. Implementaremos as duas estratégias ao mesmo tempo, para demonstração.

```
*** Test Cases ***
Scenario: Create book
  [Tags]  B006

  # Autenticação
  &{auth_payload} =   Create Dictionary  username=${API_USER}
password=${API_PASS}
  ${auth_response} =  POST  ${API_HOST}/auth  json=&{auth_payload}
expected_status=200
  ${token} =          Set Variable  ${auth_response.json()}[token]

  # Geração do novo ISBN
  ${isbn} =           FakerLibrary.Isbn13

  # Criação do novo livro
```

```
  ${body} =           Load Json From File
resources/payloads/create_book.json
  Set To Dictionary   ${body}  isbn=${isbn}
  &{header} =         Create Dictionary  x-session=${token}  Content-
Type=application/json
  ${response} =       POST  ${API_HOST}/books  json=${body}  headers=&
{header}  expected_status=201
  ${inserted_id} =    Set Variable  ${response.json()}[id]

  # Validação da entrada do novo livro na base de dados
  Should Not Be Empty  ${inserted_id}
  Open database connection
  Check If Exists In Database  SELECT isbn FROM books WHERE id =
'${inserted_id}' and isbn = '${isbn}';

  # Limpando os dados inseridos no banco de dados.
  Execute SQL String           DELETE FROM books WHERE id =
${inserted_id};
```

Nosso teste agora funciona, mas os comentários que utilizamos escancaram a necessidade de refatoração e melhor organização do código. Reescrevendo os passos com a sintaxe Gherkin e abstraindo os comandos em palavras-chave, temos o seguinte resultado:

```
*** Settings ***
Documentation  Bookstore Demo
          ...  Books Creation API Contract Testing
Library        RequestsLibrary
Library        DatabaseLibrary
Library        FakerLibrary
Library        JSONLibrary
Library        Collections
Resource       ../../../resources/config.resource
Test Tags      api  books  creation

*** Test Cases ***
Scenario: Create book
  [Tags]  B006
  Given I am authenticated
```

```
    And I get a fake ISBN
   When I post a new book with the fake ISBN
   Then I should see response status code 201
    And I should see the inserted ID in response body
    And I should see the inserted book in database
    And I should remove the inserted book from database

[...]

*** Keywords ***
I am authenticated
  ${auth_payload} =   Create Dictionary  username=${API_USER}
password=${API_PASS}
  ${auth_response} =  POST  ${API_HOST}/auth  json=${auth_payload}
expected_status=200
  Set Test Variable   ${TOKEN}  ${auth_response.json()}[token]

I get a fake ISBN
  ${fake_isbn} =     FakerLibrary.Isbn13
  Set Test Variable  ${FAKE_ISBN}   ${fake_isbn}

I post a new book with the fake ISBN
  ${body} =          Load Json From File
resources/payloads/create_book.json
  Set To Dictionary  ${body}  isbn=${FAKE_ISBN}
  &{header} =        Create Dictionary  x-session=${TOKEN}  Content-
Type=application/json
  ${response} =      POST  ${API_HOST}/books  json=${body}  headers=&
{header}
  Set Test Variable  ${RESPONSE}  ${response}

I should see response status code ${expected_status}
  Status Should Be  ${expected_status}  ${RESPONSE}

I should see the inserted ID in response body
  Set Test Variable    ${INSERTED_ID}  ${RESPONSE.json()}[id]
  Should Not Be Empty  ${INSERTED_ID}

I should see the inserted book in database
```

```
  Open database connection
  Check If Exists In Database   SELECT isbn FROM books WHERE id =
'${INSERTED_ID}';
  [Teardown]                    Close database connection

I should remove the inserted book from database
  Open database connection
  Execute SQL String            DELETE FROM books WHERE id =
'${INSERTED_ID}';
  [Teardown]                    Close database connection
```

Note que a palavra-chave `I should see response status code ${expected_status}` foi repetida neste caso de teste da mesma forma que a havíamos utilizado no caso de teste de listagem de livros. Para manter a arquitetura do projeto consistente e otimizar o desenvolvimento com reutilização de código, devemos encapsular as palavras-chave em arquivos separados dentro do diretório `resources/services/`. Inseriremos a palavra-chave `I generate an ISBN` no arquivo `dataprovider.resource`, os acessos ao banco de dados que devem ser inseridos em `database.resource`, a palavra-chave `I am authenticated` em um arquivo `auth_service.resource` e as demais em `books_service.resource`, juntamente com as palavras-chaves que foram criadas diretamente no arquivo `books_list_api_test.robot`. Não se esqueça de inserir os arquivos utilizados na lista de recursos da suíte de testes.

PATCH: Editando itens

A API permite que os dados de determinado item sejam alterados. Isso pode ser feito através do método de requisição `PATCH`. Para alterar o preço de um livro, por exemplo, podemos passar os novos valores através do corpo da requisição, conforme o exemplo a seguir:

```
{
    "price": 10.00
}
```

Implementaremos o teste de edição de livros no arquivo `books_edition_api_test.robot`. Como este também é um serviço que requer autenticação, iniciaremos o script com este procedimento. Em seguida, geraremos um livro *fake* com dados randômicos da Faker Library e o inseriremos no banco de dados. Utilizando o identificador do livro criado, realizaremos uma requisição `PATCH` para alterar o preço que havia sido cadastrado. Ao final, validaremos o código de status da resposta e que o valor foi corretamente alterado em banco de dados.

Devemos utilizar o identificador para especificar qual livro deve ser editado, mas como passar essa informação para a requisição? A resposta depende da implementação de cada sistema. No caso da API da Bookstore Demo, é através de um *path parameter*. Como vimos anteriormente, o *path parameter* envia a informação na própria URL da requisição, no formato `book/{id}`.

```
*** Settings ***
Documentation  Books Edition API Contract Testing
Test Tags      api  books  edition

Resource       ../../../resources/config.resource
Resource       ../../../resources/database.resource
Resource       ../../../resources/dataprovider.resource
Resource       ../../../resources/services/auth_service.resource
Resource       ../../../resources/services/books_service.resource

*** Test Cases ***
Scenario: Edit book
  [Tags]  B011
  Given I am authenticated
    And I get a fake book
    And I insert the fake book into database
   When I change the price of the fake book to 10.00
   Then I should see response status code 200
    And I should see the price of the fake book with value 10.00 in
database
```

A implementação da palavra-chave `I change the ${field} of the book with ID ${id} to ${new_value}`, que é a responsável pela requisição `PATH book/{id}`, deve ser feita no arquivo `books_service.resource`. Os dados que substituirão dados antigos são organizados em um dicionário que será enviado no formato JSON através do corpo da requisição. O identificador do item que será alterado é enviado como *path parameter* ao final da URL.

```
I change the ${field} of the fake book to ${new_value}
  &{payload} =   Create Dictionary  ${field}=${new_value}
  &{header} =    Create Dictionary  x-session=${TOKEN}  Content-
Type=application/json
  ${response} =  PATCH  ${API_HOST}/books/${FAKE_BOOK.id}   headers=&
{header}  json=&{payload}
  Set Test Variable  ${RESPONSE}  ${response}
```

DELETE: Removendo itens

Para remover um livro do banco de dados, devemos utilizar uma requisição do tipo `DELETE`, especificando o ID do livro ou uma lista de IDs através de *query parameters*, sem necessidade de *payload*. A resposta deve conter código de status *204 No content*, afinal, após ser removido, o recurso não existirá mais.

De modo semelhante às implementações anteriores, criaremos o arquivo `books_deletion_api_test.robot` com a documentação, as *tags*, a RequestsLibrary e a importação dos recursos dos serviços de *auth* e *books* presentes na sessão de *Settings*. Escreveremos dois casos de teste, um para validar que o serviço funciona com *apenas um item* na lista para remoção e outro para validar que o serviço funciona com *uma lista de itens* para remoção. Além de estar autenticado, os testes precisam de livros que possam ser excluídos do banco de dados.

Para garantir que não queimaremos nossa massa de dados preexistente, criaremos os dados dinamicamente e os inseriremos no banco de dados para atender à pré-condição do teste. Isso será feito

com Dataprovider e Database. Em seguida, passamos a lista de identificadores dos livros que serão removidos como parâmetro para o serviço, para então validarmos que a resposta emitiu código de status *204 No content* e verificarmos que os dados de fato não estão mais no banco de dados do sistema.

Neste capítulo, escreveremos o cenário de teste *Remove book*.

```
*** Settings ***
Documentation  Bookstore Demo
          ...  Books Deletion API Contract Testing
Library        RequestsLibrary
Resource       ../../../resources/config.resource
Resource       ../../../resources/database.resource
Resource       ../../../resources/dataprovider.resource
Resource       ../../../resources/services/auth_service.resource
Resource       ../../../resources/services/books_service.resource
Test Tags   api  books  deletion

*** Test Cases ***
Scenario: Remove book
  [Tags]  B014
  Given I am authenticated
    And I get a fake book
    And I insert the fake book into database
   When I delete the fake book
   Then I should see response status code 204
    And I should not see the fake book in the database

Scenario: Remove a books list
  [Tags]  B013
  No Operation
```

Algumas palavras-chaves como `I am authenticated`, `I get a fake book`, `I insert the fake book into database` e `I should see response status code ${expected_status}` foram reaproveitadas do código-fonte que havíamos escrito anteriormente.

A palavra-chave `I delete the fake book` pode ser implementada no arquivo `books_service.resource`:

```
I delete the fake book
  &{header} =          Create Dictionary  x-session=${TOKEN}  Content-
Type=application/json
  &{query_params} =    Create Dictionary  id=${FAKE_BOOK.id}
  ${response} =        DELETE  ${API_HOST}/books  params=&{query_params}
headers=&{header}
  Set Test Variable  ${RESPONSE}  ${response}
```

Se tudo ocorrer bem, a resposta da requisição trará o código de status *204 No content* e os dados não estarão mais presentes na base de dados. Podemos validar isso com a palavra-chave `I should not see the fake book in the database` no arquivo `database.resource`.

```
[...]

I should not see the fake book in the database
  Open database connection
  Check If Not Exists In Database  SELECT isbn FROM books WHERE id =
'${FAKE_BOOK.id}';
  [Teardown]  Close database connection
```

Assim, escrevemos alguns testes para acessar, criar, alterar e remover dados através de uma API de serviços REST. Com os conhecimentos obtidos no estudo deste livro, será possível implementar todos os cenários de teste de contrato da tabela proposta.

5.2 Testes funcionais ou de ponta a ponta

Esse tipo de teste consiste em garantir que a funcionalidade apresenta o comportamento esperado quando realizarmos uma sequência de requisições que simulam o comportamento do usuário real, e não somente em validar o código de status das respostas das

requisições, os formatos das mensagens ou tratamento de erros. Uma sequência de requisições e respostas representará um fluxo real que seria executado através da interface. Armazenaremos esses *scripts* no diretório `tests/api/e2e/` .

Um exemplo de fluxo *end-to-end* é a tentativa de adicionar um produto fora de estoque ao carrinho, que começa com a busca por um produto através do ISBN na barra de pesquisas da página inicial e termina com a verificação de que o produto não foi adicionado ao carrinho de compras. Nesse fluxo, o usuário se encontra com o carrinho de compras vazio e deseja comprar um livro que está fora de estoque. O primeiro passo é buscar o livro na lista de livros, através do seu ISBN. Como o ISBN é único, caso o livro esteja cadastrado no sistema, a lista terá somente um item e o usuário acessará as informações deste item. No entanto, ao tentar adicionar este livro ao carrinho de compras, o usuário não deve conseguir; o serviço deve emitir uma mensagem de que não foi possível prosseguir porque o livro está fora de estoque e o carrinho deve continuar vazio.

Essa lista de passos poderia descrever o comportamento de um usuário interagindo com alguma interface, mas também podemos reproduzir este fluxo diretamente através dos serviços que seriam utilizados.

```
*** Settings ***
Documentation  Stock E2E API Testing
Test Tags      api  e2e  stock

Resource       ../../../resources/config.resource
Resource       ../../../resources/database.resource
Resource       ../../../resources/services/books_service.resource
Resource       ../../../resources/services/cart_service.resource

*** Test Cases ***
Scenario: Product out of stock
  Given I get an out of stock book
    And I have an empty shopping cart
```

```
   When I search for the out-of-stock book by ISBN
    And I choose the first book of search list
    And I get the out-of-stock book
    And I add the out-of-stock book to the shopping cart
   Then I should see the fail message "Product out of stock"
    And I should not see any product in my shopping cart
```

As palavras-chaves `I get an out of stock book` e `I have an empty shopping cart` prepararão a massa de dados para o teste diretamente no banco de dados do sistema. Elas devem ser implementadas no arquivo `database.resource`.

As palavras-chaves `I search for the out-of-stock book by ISBN` e `I get the out-of-stock book` vão para o arquivo `books_service.resource`. A palavra-chave `I choose the first book of search list` será implementada neste mesmo arquivo, pois está relacionada ao serviço `/books`.

Por fim, o serviço de adicionar um item ao carrinho e suas validações deve estar em um arquivo dedicado a este *endpoint*, ao qual podemos chamar de `cart_service.resource`.

```
*** Settings ***
Library  Collections

*** Keywords ***
I add the out-of-stock book to the shopping cart
  ${payload} =         Create Dictionary  item=${OUT_OF_STOCK_BOOK.id}
  ${response} =        POST  ${API_HOST}/cart/${CART_ID}  json=${payload}
expected_status=201
  Set Global Variable  ${RESPONSE}  ${response.json()}

I should see the fail message "${expected_message}"
  Should Be Equal  ${expected_message}  ${RESPONSE}[message]

I should not see any product in my shopping cart
  ${response} =    GET  ${API_HOST}/cart/${CART_ID}  expected_status=200
  Should Be Empty  ${response.json()}[items]
```

Planejando com capricho e cuidado, rapidamente poderemos escrever novos cenários de teste apenas reaproveitando palavras-chave que já estão prontas. Isso torna a escrita de testes muito mais fácil e rápida, além de simplificar eventuais alterações no projeto.

5.3 Testes em fila de mensagens

Outra forma de comunicação entre componentes de um sistema é a mensageria, algo muito comum em aplicações baseadas em microsserviços, em que se utilizam filas de mensagens que serão lidas assincronamente por outros componentes. Os testes para essas estruturas possuem características específicas que exigem alguma compreensão sobre o funcionamento dessa abordagem arquitetural. Para estudar como funcionam essas implementações, sugiro os cursos da formação *Mensageria com Apache Kafka*, da Alura, ou o livro *Apache Kafka e Spring Boot: Comunicação assíncrona entre microsserviços*, de Eduardo Felipe Zambom Santana, publicado pela Casa do Código.

Uma das principais ferramentas do mercado para este tipo de solução é o Apache Kafka, que se define como uma plataforma distribuída de *streaming* de eventos e de código aberto. Existe uma biblioteca do Robot Framework que nos permite interagir com o Kafka para testarmos estes componentes, a KafkaLibrary. A documentação da biblioteca está em https://rawgit.com/s4int/robotframework-KafkaLibrary/master/doc/KafkaLibrary.html.

Em nosso sistema da livraria virtual, existe uma fila de mensagens implementada com Kafka para disparo de e-mails. Vejamos como ela funciona para que possamos pensar em uma solução de testes automatizados para estes componentes.

Existem distintos momentos nos quais o sistema deve enviar um e-mail para um cliente, por exemplo: quando o cliente concretiza uma compra, deve receber um e-mail de confirmação; no dia de aniversário do cliente, o sistema envia felicitações e um cupom de desconto; quando um produto volta ao estoque após o cliente tê-lo marcado como "avise-me quando disponível"; quando o cliente solicita recuperação de senha, dentre outros.

Como a demanda é alta e os serviços de disparo de e-mails são instáveis, optou-se por trabalhar com filas para evitar gargalos e garantir que nenhuma mensagem seja perdida, mesmo que seja enviada assincronamente. Alguns componentes do sistema podem inserir um item na fila de e-mails que devem ser disparados, contendo todas as informações necessárias para que isto seja feito, enquanto um serviço de disparo de e-mails fica monitorando a fila para dispará-los. Caso o serviço de disparo esteja momentaneamente fora do ar, as mensagens se acumularão na fila, mas não serão perdidas. Quando o serviço for reestabelecido, ele realizará os disparos normalmente.

A seguir, um esquema simplificado da arquitetura dessa funcionalidade:

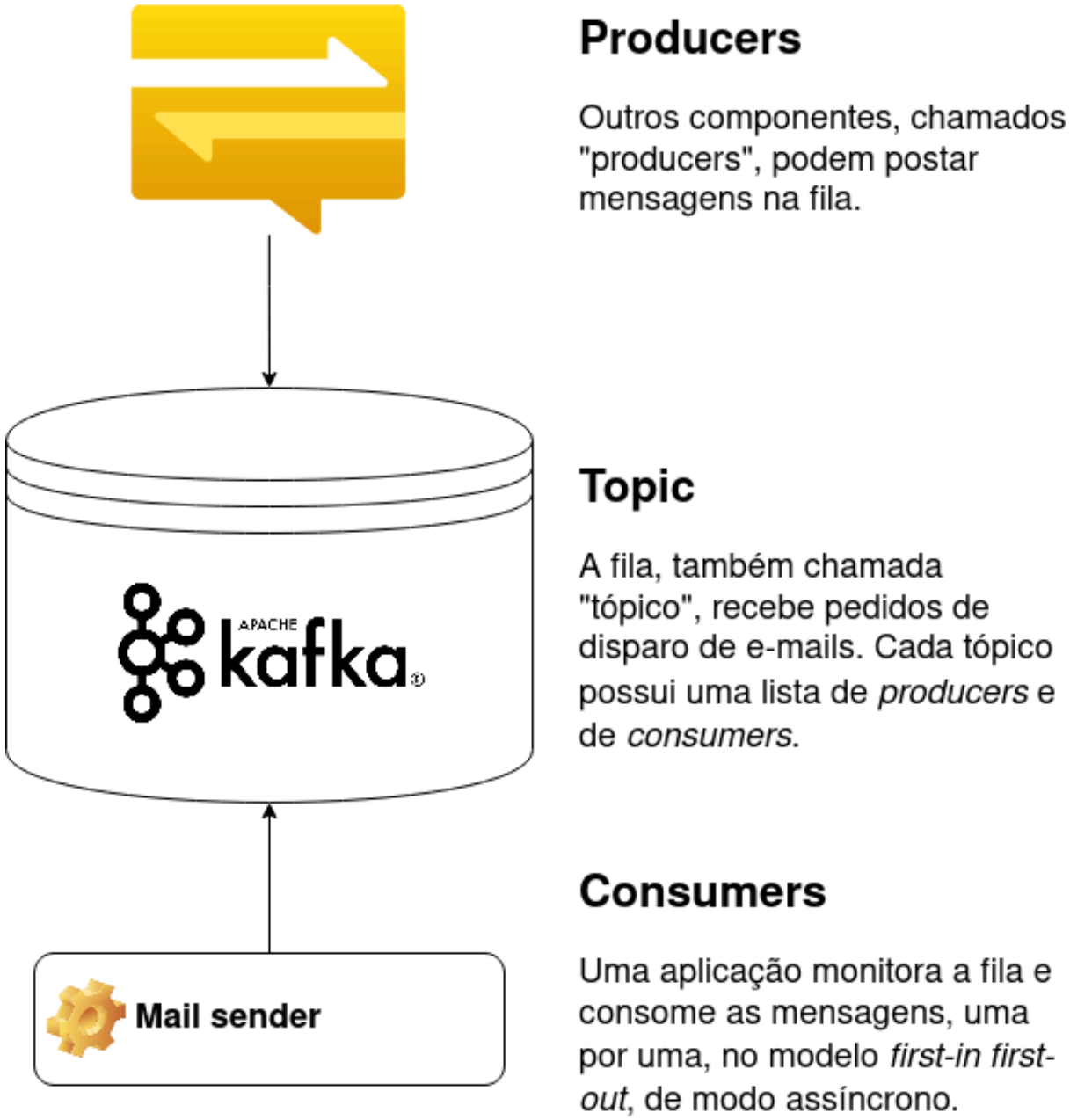


Figura 5.5: Componentes da funcionalidade de envio de e-mails.

Nossa suíte de testes será construída para validar o funcionamento do serviço `POST /reset-password`, que, ao ser invocado, postará um item na fila para envio de um e-mail para recuperação de senha, e o funcionamento do serviço de disparo de e-mails, que é responsável por ler as mensagens da fila e realizar o disparo. Para postar ou consumir itens do tópico do Kafka, utilizaremos a biblioteca

KafkaLibrary. Para validar que os e-mails estão sendo corretamente disparados, utilizaremos a biblioteca ImapLibrary.

Para utilização do serviço `POST /reset-password`, devemos realizar autenticação na API, como foi feito nos serviços anteriores, e então enviar os dados necessários para o disparo do e-mail. Os dados obrigatórios são:

- `category`: serve para definir qual tipo de mensagem deve ser enviada — neste caso, é uma mensagem de recuperação de senha;
- `name`: o nome do usuário que receberá o e-mail;
- `recipientEmailAddress`: o endereço de e-mail do cliente, para o qual deve ser disparado o e-mail;
- `passwordRecoverCode`: um código randômico que será utilizado pelo cliente para recuperar a senha.

Um exemplo de *payload* seria o JSON a seguir:

```
{
    "category": "resetPassword",
    "name": "Ana Terra",
    "recipientEmailAddress": "ana.terra@rothmail.com",
    "passwordRecoverCode": "ade37203-db76-47ae-9723-a81e2d7674a0"
}
```

Criaremos uma suíte de testes Robot chamada `mail_sender_test.robot`, na qual implementaremos os cenários de teste *Post message into the queue* e *Read from queue and send mail*. Ao criar o arquivo, podemos escrever uma breve descrição no comando `Documentation`, inserir algumas *tags* e importar alguns arquivos auxiliares que sabemos que serão utilizados, como o `config.resource` e o `dataprovider.resource`. No arquivo `config.resource`, devemos inserir o endereço do servidor do Kafka, por exemplo, `${KAFKA_SERVER} 127.0.0.1:9092`.

Podemos preparar a seção `Variables` da suíte para receber as informações relativas à caixa de e-mails de teste que utilizaremos

para validar o disparo de e-mails, sendo elas o endereço do servidor de e-mails IMAP, usuário e senha. Para os exemplos deste livro, criei uma conta para testes no Gmail, cujo endereço do servidor é `imap.gmail.com`.

```
*** Settings ***
Documentation  Bookstore Demo: E-mail sender and Kafka mail queue.

Resource       ../../resources/config.resource
Resource       ../../resources/dataprovider.resource

Test Tags       mail-sender  kafka

*** Variable ***
${TO}            test_client_bookstore_demo@gmail.com

${MAILBOX_HOST}  imap.gmail.com
${MAILBOX_USER}  test_client_bookstore_demo@gmail.com
${MAILBOX_PASS}  SenhaDaCaixaDeEmails

*** Test Cases ***
Scenario: Post message into the queue
  No Operation

Scenario: Read from queue and send mail
  No Operation
```

Para escrevermos a história de usuário em Gherkin, como fizemos nos testes anteriores, vamos pensar no passo a passo que nosso robô precisará percorrer para cumprir o objetivo. Partindo do cenário *Post message into the queue*, precisamos preparar massa de dados, ou seja, obter um nome para o cliente e um código para recuperação de senha. Geraremos estes dados randomicamente, através da Faker Library, no arquivo `dataprovider.resource`.

```
[...]

I get a fake name
  ${name} =          FakerLibrary.Name
```

```
  Set Test Variable  ${FAKE_NAME}  ${name}

I get a fake code
  ${code} =          FakerLibrary.Uuid4
  Set Test Variable  ${FAKE_CODE}   ${code}
```

Também é uma pré-condição do teste que seja feita a autenticação na API de serviços do sistema, obtendo o *token* para então realizar a requisição ao serviço `POST /reset-password`, responsável pela redefinição da senha e por postar uma mensagem na fila do Kafka com as informações para o disparo do e-mail com o código de recuperação. Faremos a autenticação através da palavra-chave já implementada anteriormente `I am authenticated`, que está no arquivo `auth_service.resource`, o qual precisará ser inserido na lista de *resources* na seção de `Settings`.

Com as pré-condições implementadas, a história de usuário deve estar assim:

```
[...]

Scenario: Post message into the queue
  Given I get a fake name
    And I get a fake code
    And I am authenticated
```

Devemos implementar agora a ação principal do teste, que é invocar o serviço `POST /reset-password`. Implementaremos uma palavra-chave para essa tarefa dentro de um arquivo que abstrairá esse serviço, o qual podemos chamar de `password_service.resource`, no diretório `/resources/services`.

A requisição possui um *payload*, que será montado em um dicionário contendo as informações no formato JSON, como fizemos anteriormente nos testes de contrato. Como esse serviço posta a mensagem na fila para que o e-mail seja disparado posteriormente, o código de status HTTP retornado na resposta será *202 Accepted*, que significa que a requisição foi aceita, mas ainda não foi

processada. Em outras palavras, informa que a requisição será processada assincronamente.

Perceba que a variável `${TO}`, ao ser declarada na seção *Variable* da Test Suite, pode ser utilizada diretamente dentro da palavra-chave implementada no arquivo de *resource* porque as variáveis ali declaradas recebem automaticamente definição de escopo "SUITE".

```
*** Settings ***
Library  RequestsLibrary
Library  Collections
Library  JSONLibrary

*** Variable ***
${MESSAGE_TYPE}   passwordRecover

*** Keywords ***
I ask for password reset
  &{header} =    Create Dictionary  x-session=${TOKEN}            Content-
Type=application/json
  &{payload} =    Create Dictionary  name=${FAKE_NAME}
recipientEmailAddress=${TO}
                                ...  category=${MESSAGE_TYPE}
passwordRecoverCode=${FAKE_CODE}
  ${response} =   POST               ${API_HOST}/reset-password   json=&
{payload}  headers=&{header}  expected_status=202
  Set Test Variable  ${RESPONSE}  ${response}
```

Agora precisamos monitorar o tópico do Kafka para verificar se o serviço postará a mensagem corretamente. Então, realizaremos a conexão com o servidor do Kafka, onde devemos definir um `client_id` e um `group_id`, que podem ser valores arbitrários. Logo em seguida, o Robot deve se cadastrar como consumidor para escutar os eventos do tópico, quando informaremos o nome do tópico onde estão as mensagens que nos interessam, neste caso `mailQueue`. Implementaremos esses passos utilizando recursos da KafkaLibrary, em um arquivo *resource* separado, com a

responsabilidade única de interagir com o servidor Kafka. Nomearemos este arquivo de `kafka.resource`.

```
*** Settings ***
Documentation  Bookstore Demo: Kafka resources.
Library        KafkaLibrary
Library        String

*** Variable ***
${ENCODING}  UTF-8
${GROUP_ID}  Robot

*** Keywords ***
I connect to Kafka as Consumer
  Connect To Kafka     bootstrap_servers=${KAFKA_SERVER}
auto_offset_reset=latest    client_id=RobotConsumer
group_id=${GROUP_ID}
  Connect Consumer     bootstrap_servers=${KAFKA_SERVER}
client_id=RobotConsumer     group_id=${GROUP_ID}
enable_auto_commit=${TRUE}

I connect to Kafka as Producer
  Connect To Kafka     bootstrap_servers=${KAFKA_SERVER}
auto_offset_reset=latest    client_id=RobotProducer
group_id=${GROUP_ID}
  Connect Producer     bootstrap_servers=${KAFKA_SERVER}
client_id=RobotProducer     group_id=${GROUP_ID}

I subscribe ${topic} topic
  Subscribe Topic   ${topic}
```

Com isso, nosso caso de teste deve estar parecido com este:

```
[...]

Scenario: Post message into the queue
  Given I get a fake name
    And I get a fake code
    And I am authenticated
   When I ask for password reset
```

```
    And I connect to Kafka as Consumer
    And I subscribe mailQueue topic
```

Resta-nos implementar as validações das pós-condições do teste. Faremos uma série de verificações: devemos validar o código de status da resposta do serviço, validar que uma nova mensagem foi postada na fila, validar que esta mensagem possui os campos `id` e `datetime` e validar que os campos `name`, `passwordRecoverCode` e `recipientEmailAddress` conferem com os valores que foram enviados. Como essas ações se referem a interações com o Kafka, vamos implementá-las no arquivo `kafka.resource`.

O comando `Poll` retornará a próxima mensagem da fila, cujo conteúdo estará no formato *bytes*. Esse conteúdo precisará ser convertido para um objeto JSON para que possamos trabalhar com ele nas validações. Para validar os campos, podemos criar duas palavras-chaves parametrizadas para que sejam reaproveitadas para distintos campos do objeto JSON. São elas: `I should see the ${field_name} field` e `I should see the ${field_name} field equals to ${expected_value}`.

```
[...]

I should see a new message on the queue
  ${messages} =            Poll  max_records=${1}  timeout_ms=${500}
  ${message_as_string} =   Decode Bytes To String  ${messages[0].value}
${ENCODING}
  ${message_as_json} =     Convert String To Json  ${message_as_string}
  Set Test Variable        ${KAFKA_LAST_MESSAGE}   ${message_as_json}

I should see the ${field_name} field
  Should Not Be Empty  ${KAFKA_LAST_MESSAGE}[id]

I should see the ${field_name} field equals to ${expected_value}
  Should Be Equal      ${KAFKA_LAST_MESSAGE}[${field_name}]
${expected_value}
```

É muito importante lembrar que a conexão com o servidor do Kafka também precisa ser encerrada ao final da execução. Para isso,

devemos utilizar o comando `Test Teardown` para executar a palavra-chave `Close` da KafkaLibrary, que garantirá o encerramento seguro da conexão independentemente do resultado do teste. O código final do caso de teste ficou assim:

```
*** Settings ***
Documentation    Bookstore Demo: Kafka Mail queue.

Resource         ../../resources/config.resource
Resource         ../../resources/kafka.resource
Resource         ../../resources/dataprovider.resource
Resource         ../../resources/services/auth_service.resource
Resource         ../../resources/services/password_service.resource

Test Tags        mail-sender  kafka
Suite Teardown   Run Keyword And Ignore Error    Close

*** Variable ***
${TO}            test_client_bookstore_demo@gmail.com

${MAILBOX_HOST}  imap.gmail.com
${MAILBOX_USER}  test_client_bookstore_demo@gmail.com
${MAILBOX_PASS}  SenhaDaCaixaDeEmails

*** Test Cases ***
Scenario: Post message into the queue
  Given I get a fake name
    And I get a fake code
    And I am authenticated
   When I ask for password reset
    And I connect to Kafka as Consumer
    And I subscribe mailQueue topic
   Then I should see response status code 202
    And I should see a new message on the queue
    And I should see the id field
    And I should see the datetime field
    And I should see the name field equals to ${FAKE_NAME}
    And I should see the passwordRecoverCode field equals to ${FAKE_CODE}
    And I should see the recipientEmailAddress field equals to ${TO}
```

No cenário seguinte, *Read from queue and send mail*, o robô postará uma mensagem diretamente na fila de e-mails, sem utilizar a API de serviço. Assim, buscamos garantir o isolamento do componente *Mail Sender*, que é o alvo deste caso de teste. Após a inserção da mensagem no tópico do Kafka, verificaremos se o e-mail foi disparado corretamente, validando-o diretamente na caixa de e-mails.

O e-mail que deve ser enviado para o usuário deverá conter o código de recuperação de senha, como na imagem a seguir:

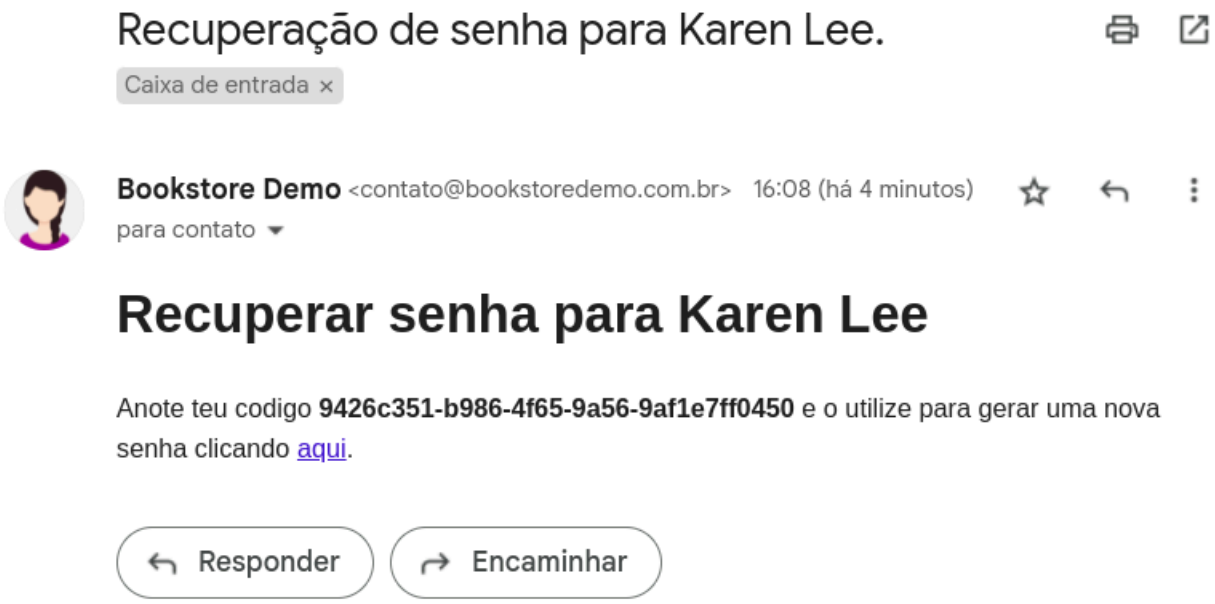


Figura 5.6: Exemplo de uma mensagem de e-mail enviada pelo sistema.

As pré-condições deste caso de teste também incluem a geração randômica do nome do cliente e do código de recuperação da senha, e também de um identificador único arbitrário para a mensagem e de um *timestamp*. Utilizaremos novamente o Faker para gerar estes dados, dentro do nosso Dataprovider.

```
[...]

I get a fake id
  ${id} =        FakerLibrary.RandomNumber  digits=4
  Set Test Variable  ${FAKE_ID}  ${id}
```

```
I get a fake datetime
  ${datetime} =  FakerLibrary.Date
  Set Test Variable  ${FAKE_DATETIME}  ${datetime}
```

Organizaremos os dados em um dicionário, que representará a mensagem que será postada no tópico, contendo todas as informações necessárias para o disparo do e-mail. Por ser uma palavra-chave apenas auxiliar, que servirá para não quebrar a legibilidade do cenário e que não se encaixa nas responsabilidades dos outros componentes, optei por implementá-la diretamente no arquivo da suíte de teste, na seção `Keywords`.

```
[...]

*** Keywords ***
I get a dictionary with password recover data
  &{data} =  Create Dictionary  id=${FAKE_ID}
category=passwordRecover
                            ...  name=${FAKE_NAME}
recipientEmailAddress=${TO}
                            ...  datetime=${FAKE_DATETIME}
passwordRecoverCode=${FAKE_CODE}
  Set Global Variable           ${PASSWORD_RECOVER_DATA}     &{data}
```

Realizaremos a conexão com o Kafka e a subscrição como *producer*, nos habilitando para postar mensagens na fila. O envio de mensagem para o tópico será implementado no componente `kafka.resource`, evidentemente. A mensagem deve ser convertida para *bytes* antes de ser postada. É importante não esquecer do comando `Flush` ao final para consolidar o envio da mensagem.

```
[...]

I send a message to ${topic} topic with password recover data
  ${message_as_string} =  Convert Json To String
${PASSWORD_RECOVER_DATA}
  ${bytes} =              Encode String To Bytes    ${message_as_string}
${ENCODING}
```

```
  Send                    ${topic}                value=${bytes}
  Flush
```

Assim, a primeira parte do caso de teste está pronta:

```
[...]

Scenario: Read from queue and send mail
  Given I get a random name
    And I get a random code
    And I get a random id
    And I get a random datetime
    And I get a dictionary with password recover data
    And I connect to Kafka as Producer
    And I send a message to mailQueue topic with password recover data
```

Validando o disparo de e-mails

Agora precisamos abrir a caixa de e-mails e monitorá-la, aguardando a chegada da mensagem de e-mail com o código de recuperação de senha que deve ser enviada pelo *Mail Sender*. Implementaremos as interações com a caixa de e-mails no arquivo `email.resource`, através da biblioteca *ImapLibrary2*.

A biblioteca ImapLibrary2 é um *fork* da biblioteca ImapLibrary (https://github.com/rickypc/robotframework-imaplibrary), mas que não é mantida atualizada desde 2016. Esse é o motivo da escolha pela ImapLibrary2, cuja última versão foi gerada em novembro de 2022.

Essa biblioteca implementa o protocolo IMAP, o qual, como vimos no início deste capítulo, é um protocolo de gerenciamento de correio eletrônico. A biblioteca ImapLibrary2 utiliza esse protocolo para que o Robot Framework possa acessar uma caixa de e-mails, monitorar a chegada de novas mensagens e analisar seu conteúdo, sem a necessidade de interface gráfica.

Para abrir a caixa de e-mails, passamos como argumentos o servidor de e-mails, nome de usuário e senha da conta de e-mail que criamos

para teste. Alguns servidores de e-mails bloqueiam este tipo de acesso, mas é possível acessar as configurações da conta para configurar e liberar o acesso IMAP.

Para monitorar a entrega da mensagem de e-mail que estamos esperando, utilizamos a palavra-chave `Wait for email`, realizando alguns filtros. Existem vários filtros possíveis, que estão descritos na documentação das palavras-chave da biblioteca. Optei por utilizarmos o remetente e o assunto da mensagem, que são suficientes para garantir que receberemos a mensagem correta.

Precisamos definir também o parâmetro `poll_frequency`, em segundos, que representa o intervalo de tempo a cada nova tentativa em que o robô verificará se a mensagem chegou, e o parâmetro `timeout`, também em segundos, que representa o tempo limite de espera até que a mensagem chegue antes de o teste falhar. Com estes parâmetros definidos na seção `Variables` e passados como argumentos na chamada da palavra-chave `Wait for email`, o robô verificará a cada 5 segundos se a mensagem chegou, e, caso não chegue em até 30 segundos, o teste falhará.

```
*** Settings ***
Library        ImapLibrary2

*** Variables ***
${POLL_FREQUENCY}  5
${TIMEOUT}         30

*** Keywords ***
I open the mail box
  Open Mailbox  host=${MAILBOX_HOST}  user=${MAILBOX_USER}
password=${MAILBOX_PASS}

I wait for e-mail message
  ${subject} =   Set Variable    Recuperar senha para ${FAKE_NAME}
  ${latest} =    Wait For Email  sender=${MAILBOX_USER}
subject=${subject}   poll_frequency=${POLL_FREQUENCY}  timeout=${TIMEOUT}
  Set Test Variable  ${LATEST_MAIL}        ${latest}
```

Ao identificar que a mensagem de e-mail chegou à caixa de e-mails, acessamos seu conteúdo para validar que as informações contidas nele estão coerentes com o que foi solicitado. Realizaremos isso através da palavra-chave `Get email body`, em implementação dentro do arquivo `email.resource`. Existem outras possibilidades de validação disponíveis dentro da biblioteca ImapLibrary que podem ser exploradas, como a validação de links.

```
[...]

I should see the fake code in e-mail body
  ${html} =       Get Email Body  ${LATEST_MAIL}
  Should Contain  ${html}         Anote teu codigo <strong>${FAKE_CODE}
</strong>
```

Ao final do teste, devemos fechar a caixa de e-mails, utilizando o comando `Teardown` do caso de teste. Criamos uma palavra-chave para essa tarefa no arquivo `email.resource`, onde estão todas as palavras-chave relacionadas ao monitoramento de caixa de e-mails.

```
[...]

I close the mail box
  Run Keyword And Ignore Error  Close Mailbox
```

Dessa forma, temos tudo que precisamos para concluir mais este cenário de teste, o segundo da suíte de testes para a funcionalidade de disparo de e-mails do sistema. A seguir, a versão completa do caso de teste que acabamos de implementar.

```
*** Settings ***
Documentation  Bookstore Demo: Kafka Mail queue.

Resource       ../../resources/config.resource
Resource       ../../resources/email.resource
Resource       ../../resources/kafka.resource
Resource       ../../resources/dataprovider.resource
Resource       ../../resources/services/auth_service.resource
Resource       ../../resources/services/password_service.resource
```

```
Test Tags       kafka
Suite Teardown   Run Keyword And Ignore Error    Close

*** Variable ***
${TO}            test_client_bookstore_demo@gmail.com

*** Test Cases ***
Scenario: Post message into the queue
  [Tags]  post
  Given I get a fake name
    And I get a fake code
    And I am authenticated
   When I ask for password reset
    And I connect to Kafka as Consumer
    And I subscribe mailQueue topic
   Then I should see response status code 202
    And I should see a new message on the queue
    And I should see the id field
    And I should see the datetime field
    And I should see the name field equals to fake name
    And I should see the passwordRecoverCode field equals to fake ode
    And I should see the recipientEmailAddress field equals to recipient
e-mail

Scenario: Read from queue and send mail
  [Tags]  read
  Given I get a fake name
    And I get a fake code
    And I get a fake id
    And I get a fake datetime
    And I get a dictionary with password recover data
    And I connect to Kafka as Producer
    And I send a message to mailQueue topic with password recover data
   When I open the mail box
    And I wait for e-mail message
    And I should see the fake code in e-mail body
    [Teardown]  I close the mail box
```

```
*** Keywords ***
I get a dictionary with password recover data
  &{data} =  Create Dictionary  id=${FAKE_ID}
category=passwordRecover
                              ...  name=${FAKE_NAME}
recipientEmailAddress=${TO}
                              ...  datetime=${FAKE_DATETIME}
passwordRecoverCode=${FAKE_CODE}
  Set Test Variable             ${PASSWORD_RECOVER_DATA}  &{data}
```

Como podemos notar, os testes de API são simples de construir e rápidos de executar, mas é preciso estruturá-los consistentemente para que o código não se torne confuso. Por trás dos testes implementados nesta camada, existirão conceitos específicos de arquitetura relativos ao software que será testado. Por isso, encorajo o estudo complementar sobre APIs de serviços, mensageria e outras formas de arquitetura de software. Relembrando o que estudamos na introdução deste livro sobre a pirâmide de testes, esta é a camada de testes automatizados que gerará maior custo-benefício ao time.

CAPÍTULO 6
Estruturas de controle

Eu realmente procuro evitar utilizar estruturas de controle clássicas em meus projetos com Robot Framework, pois, apesar da sintaxe simplificada do Robot, resolver tudo utilizando somente palavras-chaves torna o código mais simples. No entanto, há ocasiões em que recursos mais complexos são necessários.

Retomaremos a suíte de testes do serviço de remoção de livros, o `DELETE /books`, no qual implementamos apenas um teste para o cenário de exclusão de um único item. Implementaremos agora o caso de teste *Remove a books list*, através do qual validaremos a possibilidade de remover uma lista de livros em uma única requisição. Os passos a serem executados são parecidos com o caso de teste anterior:

```
[...]

*** Test Cases ***
Scenario: Remove book
  [Tags]  B014
  Given I am authenticated
    And I get a fake book
    And I insert the fake book into database
   When I delete the fake book
   Then I should see response status code 204
    And I should not see the fake book in the database

Scenario: Remove a books list
  [Tags]  B013
  Given I am authenticated
    And I get 3 fake books
    And I insert all fake books into database
   When I delete the fake books
```

```
   Then I should see response status code 204
    And I should not see the fake book list in the database
```

Observando os cenários descritos, percebe-se que são bastante semelhantes e parecem muito simples. Contudo, precisamos de alguns recursos interessantes da ferramenta para que este script funcione. Neste cenário, precisaremos de uma lista de livros como massa de dados, em vez de apenas um único livro, como foi suficiente até então.

O que precisamos fazer, na prática, é executar *n* vezes a palavra-chave `I get a fake book` do Dataprovider para obter uma lista de livros *fake* e, logo depois, inserir cada item da lista de livros *fake* no banco de dados. Estas são duas tarefas clássicas que podem ser resolvidas através de *laços de repetição*.

6.1 Estruturas de repetição

O Robot Framework possui as estruturas `FOR` e `WHILE`, e a sintaxe é baseada na linguagem Python, de modo que se assemelha também à maioria das linguagens de programação de mercado.

Para executar um conjunto de comandos *n* vezes, como precisaremos fazer em nosso Dataprovider, utilizaremos um laço de repetição capaz de iterar *n* vezes, o `FOR IN RANGE`. Com ele, criaremos *n* livros *fake* que serão guardados em uma lista *fake* de livros.

```
[...]

I get ${qtd} fake books
  @{fake_book_list} =  Create List
  FOR  ${index}  IN RANGE  ${qtd}
    &{fake_book} =  I get a fake book
    Append To List  @{fake_book_list}   ${fake_book}
```

```
  END
  Set Test Variable  @{FAKE_BOOK_LIST}   @{fake_book_list}
```

Agora que temos uma lista de livros para inserir em banco de dados, precisamos iterar sobre esta lista para inserir um livro de cada vez. O laço de repetição `FOR IN` é capaz de iterar sobre cada item de uma lista e resolve nosso problema. A palavra-chave `I insert all ${books_list} into database` simplesmente receberá uma lista de livros e, para cada um deles, chamará a palavra-chave `I insert ${book} into database` que já havia sido implementada.

```
[...]

I insert all fake books into database
  FOR  ${book}  IN  @{FAKE_BOOK_LIST}
    I insert ${book} into database
  END
```

Com a massa de dados preparada, podemos realizar a requisição `DELETE` com o livro ou com a lista de livros que preparamos. Contudo, um identificador de livro é diferente de uma lista de livros. Precisaremos alterar a forma como implementamos a palavra-chave `I delete book`, pois existem duas formas distintas para lidar com o parâmetro de entrada, dependendo deste valor. Essa tomada de decisão pode ser construída com uma *estrutura de seleção*.

6.2 Estruturas de seleção

O Robot Framework possui o recurso `IF/ELSE` para implementação de tomadas de decisão, mas é fortemente recomendável que, se possível, sejam utilizadas as palavras-chave condicionais. No entanto, esse não é nosso caso, já que não se trata de simplesmente executar uma palavra-chave condicionalmente.

Precisaremos implementar o comando `IF/ELSE` para saber se estamos lidando com o identificador de um livro, ou seja, um valor, ou se estamos lidando com vários livros, ou seja, uma lista.

Isso será necessário para preparar a lista de identificadores que serão passados para a chamada da requisição, que deve ter o formato `DELETE /books?id=101` ou `DELETE /books?id=101,102,105`. Se houver somente um livro para remover, a construção do *path parameter* fica muito simples, mas, se houver uma lista de livros, será preciso converter essa lista para uma *string* no formato adequado, com vários ids separados por vírgula. Esta tarefa será feita pela palavra-chave `Get comma separated list as string`, cuja implementação não interferirá no estudo deste capítulo e, portanto, não será detalhada aqui.

Alteraremos a declaração da palavra-chave para `I delete the fake book${s}` para que ela possa ser utilizada em ambos os casos.

Utilizaremos uma tática léxica nesta palavra-chave, através do parâmetro embutido `${s}`. Ele serve para permitir a escrita da diretiva tanto no singular quanto no plural. Caso seja escrita no singular, então o parâmetro `${s}` ficará vazio e o parâmetro que será utilizado deve ser apenas um identificador. Caso contrário, o parâmetro deve ser uma lista. Esta lista será então convertida para o formato correto e passado como um *query parameter* da maneira como já conhecemos. Para identificar se o parâmetro está vazio, compararemos seu valor com a variável padrão `${EMPTY}` do Robot Framework, a qual representa um valor vazio.

```
[...]

I delete the fake book${s}
  IF  '${s}' == '${EMPTY}'
    ${formated_param_list} =  Set Variable    ${FAKE_BOOK.id}
  ELSE
    ${formated_param_list} =  Get comma separated list as string
list=@{FAKE_BOOK_LIST}
  END
```

```
  &{header} =        Create Dictionary  x-session=${TOKEN}  Content-
Type=application/json
  &{query_params} =  Create Dictionary  id=${formated_param_list}
  ${response} =      DELETE  ${API_HOST}/books  params=&{query_params}
headers=&{header}
  Set Test Variable  ${RESPONSE}  ${response}
```

A validação de que os livros foram realmente removidos do banco de dados também precisa ser adaptada para uma lista de livros. Implementaremos a palavra-chave `I should not see the book${s} from ${books_list} in the database`, que receberá como argumento a lista de identificadores dos livros removidos no passo anterior e, para cada um dos itens da lista, verificará se ele está presente no banco de dados através da já existente palavra-chave `I should not see the book ${book.id} in the database`.

```
[...]

I should not see the fake book list in the database
  FOR  ${book}  IN  @{FAKE_BOOK_LIST}
    I should not see the book ${book.id} in the database
  END
```

6.3 Estrutura de tratamento de exceções

No capítulo anterior, criamos a palavra-chave `I get an out of stock book` dentro do componente de Database do nosso projeto. Ela serve para buscar um livro que esteja com o estoque zerado em base de dados, o qual o teste utilizará como massa de dados.

Na primeira versão que implementamos, realizávamos uma verificação com a palavra-chave `Should Not Be Empty`, que falharia caso não houvesse nenhum registro que satisfizesse nossa condição de filtro na consulta ao banco de dados. Contudo, não é exatamente isso que gostaríamos de ver nos relatórios de execução de nossa

bateria de testes. Na verdade o teste não falhou, afinal, nem foi executado.

Quando determinado caso de teste não puder ser executado devido a algum impedimento alheio ao sistema sob teste, convém forçar o Robot a "pular" este caso de teste. Isso pode ser feito através do comando `Skip`. Também é possível forçar o Robot a falhar um caso de teste, utilizando o comando `Fail`.

Vamos alterar o código da palavra-chave `I get an out of stock book` e utilizar a estrutura `TRY/EXCEPT` para tratamento de exceções. Ela funciona de forma análoga ao tratamento de exceções do Python.

Caso ocorra algum problema na sequência de comandos no escopo do `TRY`, uma exceção será lançada e capturada no escopo `EXCEPT`. É possível filtrar os tipos de exceções utilizando a mensagem que a descreve e guardar esta informação em uma variável, mas não precisaremos fazer isso neste caso no qual estamos trabalhando. Se não especificarmos nada, toda e qualquer exceção será capturada.

Quando não houver nenhum registro no resultado da consulta, nos depararemos com a mensagem de erro "*List '@{resultset}' has no item in index 0*" ao tentar acessar o primeiro item da lista, pois a lista estará vazia e nenhum item existirá. Nós precisamos explicar de forma mais clara qual problema realmente ocorreu para que o analista de teste possa compreender mais facilmente o relatório de execução.

Logs

Para facilitar o *debug* e enriquecer os relatórios de execução com informações sobre o teste, existe a palavra-chave `Log`, nativa do Robot Framework. Existem variações desta palavra-chave para casos específicos, como `Log to console`, `Log many`, `Log Dictionary`, `Log list` etc.

Podemos definir um nível de criticidade do *log* através do parâmetro `level`. Existem diferentes níveis de *logs*: FAIL, ERROR, WARN, INFO, DEBUG e TRACE.

O nível FAIL pode ser utilizado pelo próprio Robot Framework. As mensagens de nível WARN ou ERROR ficarão automaticamente visíveis também no console e na sessão *Test Execution Errors* do relatório.

Independentemente do que for executado, ao final é indispensável encerrar a conexão com banco de dados. Isso pode ser feito no escopo do `FINALLY` e funciona exatamente da mesma forma que o *keyword teardown*.

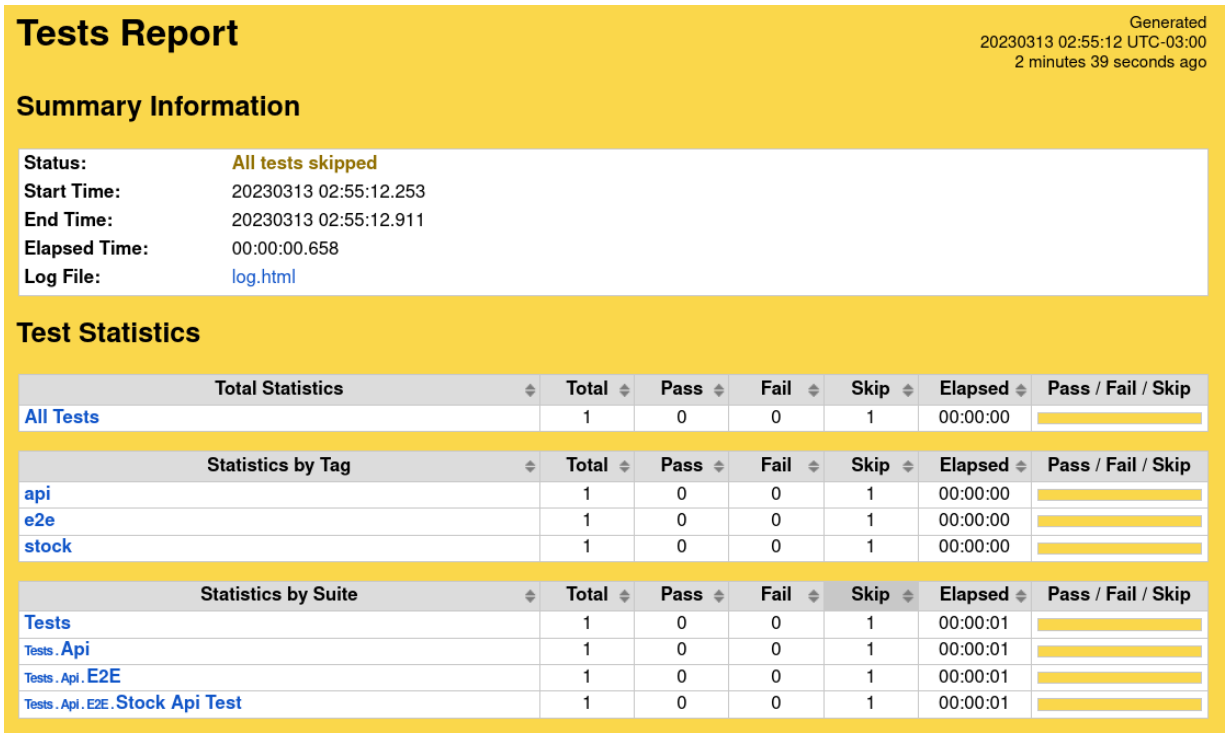
```
I get an out of stock book
  TRY
    Open database connection
    @{resultset} =     Query                 SELECT id, isbn FROM books
WHERE stock = 0 LIMIT 1;   returnAsDict=True
    &{book_item} =     Set Variable          ${resultset}[0]
    Set Test Variable  ${OUT_OF_STOCK_BOOK}  ${book_item}
  EXCEPT
    Log   None valid product found in database.     level=WARN
    Skip  Insufficient data for testing.
  FINALLY
    Close database connection
  END
```

Perceba que já no console de execução fica explícito o motivo pelo qual o teste precisou ser pulado.

```
Tests.Api.E2E.Stock Api Test :: Bookstore Demo Stock E2E API Testing
==============================================================================
[ WARN ] None valid product found in database.
Scenario: Product out of stock                                        | SKIP |
Insufficient data for testing.
------------------------------------------------------------------------------
Tests.Api.E2E.Stock Api Test :: Bookstore Demo Stock E2E API Testing  | SKIP |
1 test, 0 passed, 0 failed, 1 skipped
```

Figura 6.1: Console da execução após Skip.

Analise como ficaram os relatórios da execução. Os cenários que não puderam ser executados foram marcados em amarelo devido ao *log* de nível *Warning* que utilizamos.



Tests Report

Generated
20230313 02:55:12 UTC-03:00
2 minutes 39 seconds ago

Summary Information

Status:	All tests skipped
Start Time:	20230313 02:55:12.253
End Time:	20230313 02:55:12.911
Elapsed Time:	00:00:00.658
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	1	0	0	1	00:00:00	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
api	1	0	0	1	00:00:00	
e2e	1	0	0	1	00:00:00	
stock	1	0	0	1	00:00:00	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Tests	1	0	0	1	00:00:01	
Tests . Api	1	0	0	1	00:00:01	
Tests . Api . E2E	1	0	0	1	00:00:01	
Tests . Api . E2E . Stock Api Test	1	0	0	1	00:00:01	

Figura 6.2: Skip report.

Os logs de execução são interessantes para observar o fluxo de processamento do Robot Framework dentro da estrutura de tratamento de exceções.

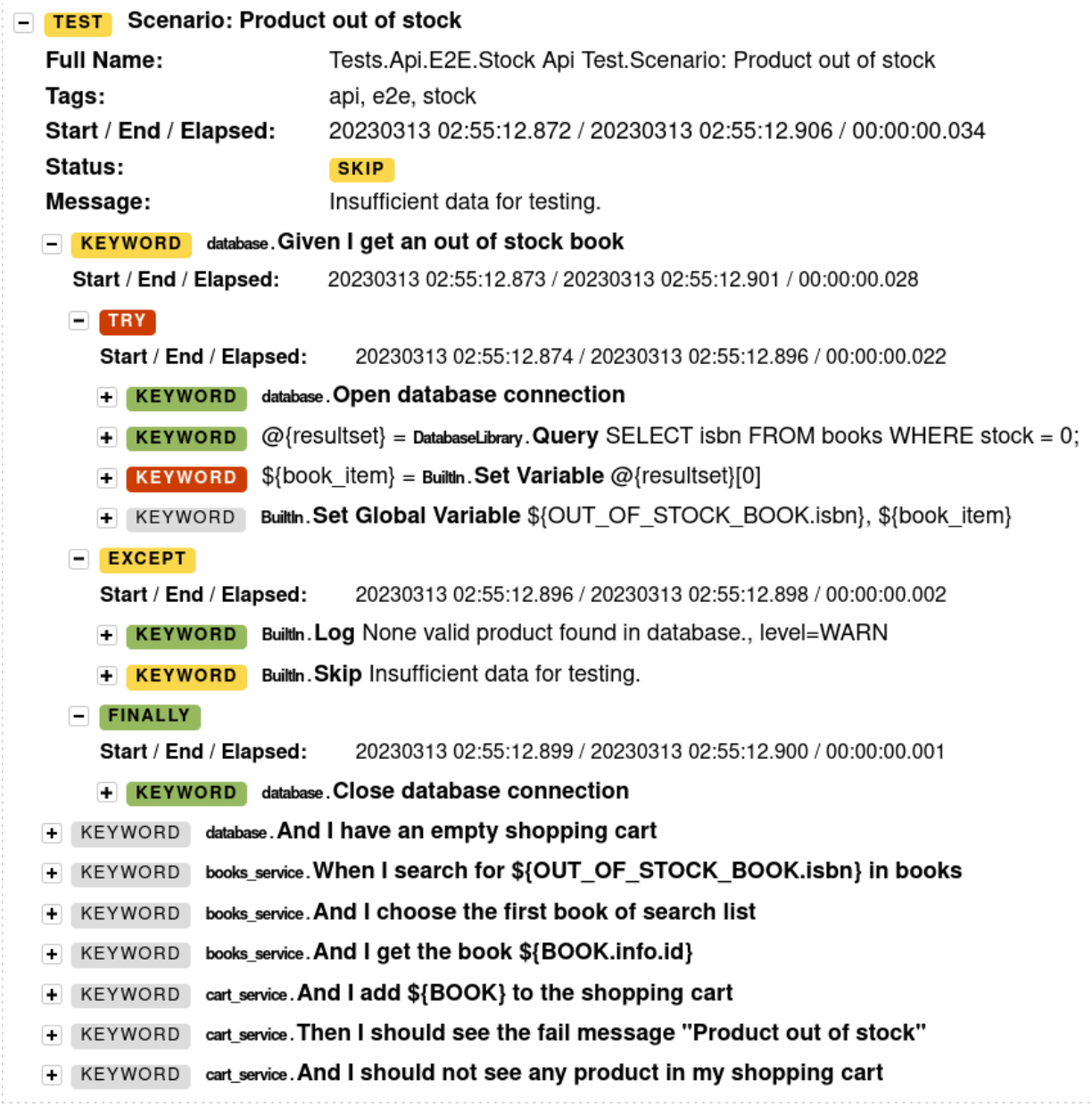


Figura 6.3: Skip logs.

Existem algumas *built-in keywords* do Robot que realizam praticamente o mesmo trabalho que o `TRY/EXCEPT` nativo. Entretanto, a recomendação do guia de usuário do Robot Framework recomenda a utilização do comando nativo, exceto quando for necessário obter compatibilidade com versões da ferramenta anteriores à 5.0.

Para conhecer mais sobre estruturas de controle do Robot Framework, como o `WHILE`, por exemplo, leia mais em https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#control-structures.

Capítulo 7
Testes para dispositivos móveis

Nosso time de desenvolvimento lançou a primeira versão de um aplicativo móvel para que os leitores possam comprar seus livros através do telefone celular, tanto no Android quanto no iOS. Esses aplicativos são sistemas de *front-end* que apenas exibem uma interface de usuário distinta, mas que consomem a mesma API de serviços que roda no *back-end* e a qual já conhecemos e testamos. Ainda assim, algumas validações precisam ser feitas através de testes diretamente na interface.

Nós podemos utilizar o Robot Framework para testar os aplicativos para dispositivos móveis, através da biblioteca Appium (documentação em https://serhatbolsu.github.io/robotframework-appiumlibrary/AppiumLibrary.html). O Appium é capaz de executar testes tanto para Android como para iOS e é uma das ferramentas mais utilizadas do mercado para essa tarefa.

7.1 Preparação e configuração

A configuração de todo ambiente de desenvolvimento para trabalhar com testes de aplicativos para dispositivos móveis com Appium costuma ser bastante trabalhosa e muitas vezes mais complexa do que a construção do próprio projeto de automação. Devido a isso, existe um programa chamado Appium Doctor, cujo objetivo é diagnosticar a correta configuração do ambiente de desenvolvimento para trabalhar com Appium. Além da AppiumLibrary, precisaremos do JavaSDK, Android SDK, Emulador Android, NodeJS, Appium Server e Appium Desktop.

Para instalar a biblioteca do Appium para Robot Framework, execute o comando a seguir:

```
$ pip install robotframework-appiumlibrary
```

Para conferir se a biblioteca foi corretamente instalada, execute o comando `pip list` e verifique se o Appium está na lista que será exibida.

Baixe o instalador do Java SDK em www.oracle.com/java/technologies/downloads/ de acordo com seu sistema operacional, realize a instalação e não esqueça de definir a variável de ambiente `$JAVA_HOME` e adicioná-la à variável `$PATH` do sistema operacional.

O Android SDK e o Emulador Android estão disponíveis para download em https://developer.android.com/studio. Após a instalação, confira se a variável de ambiente `$ANDROID_HOME` foi corretamente definida e adicionada à variável `$PATH`. Adicione também o emulador ao conteúdo da variável `$PATH` para que seja possível inicializá-lo diretamente pelo terminal, sem a necessidade de abrir o Android Studio. Na primeira vez, será necessário utilizar o gerenciador de dispositivos do Android Studio para baixar e instalar um dispositivo Android qualquer para o emulador, então será possível inicializá-lo com o comando a seguir:

```
$ emulator -avd Nexus_6
```

Baixe o NodeJS em https://nodejs.org/download e realize a instalação. Para conferir se tudo ocorreu bem, abra o terminal e execute os comandos `node --version` e `npm -v`.

Agora será possível proceder com a instalação do Appium através do NPM, o gerenciador de pacotes do NodeJS:

```
$ npm install -g appium
```

O Appium Desktop não é obrigatório, mas facilita muito. Baixe o executável em appium.io/downloads.html.

Por fim, o Appium Doctor, que realizará a verificação e diagnóstico do ambiente de desenvolvimento para Appium, pode ser instalado através do NPM:

```
$ npm install @appium/doctor -g
```

Para utilizá-lo, execute o comando `appium-doctor` passando o parâmetro `--android` ou `--ios`, de acordo com o sistema operacional alvo.

```
$ appium-doctor --android
```

Caso todos os passos tenham sido executados corretamente e tudo tenha ocorrido bem, o resultado será parecido com este:

```
info AppiumDoctor Appium Doctor v.2.0.9
info AppiumDoctor ### Diagnostic for necessary dependencies starting ###
info AppiumDoctor  ✔ APPIUM_HOME is /home/thomasvieira/.appium
info AppiumDoctor  ✔ The Node.js binary was found at: /usr/bin/node
info AppiumDoctor  ✔ Node version is 18.14.2
info AppiumDoctor  ✔ ANDROID_HOME is set to: /home/thomasvieira/Android/Sdk
info AppiumDoctor  ✔ JAVA_HOME is set to: /usr/java/jdk-17.0.4.1/
info AppiumDoctor    Checking adb, emulator, apkanalyzer
info AppiumDoctor      'adb' is in /home/thomasvieira/Android/Sdk/platform-tools/adb
info AppiumDoctor      'emulator' is in /home/thomasvieira/Android/Sdk/emulator/emulator
info AppiumDoctor      'apkanalyzer' is in /home/thomasvieira/Android/Sdk/tools/bin/apkanalyzer
info AppiumDoctor  ✔ adb, emulator, apkanalyzer exist: /home/thomasvieira/Android/Sdk
info AppiumDoctor  ✔ 'bin' subfolder exists under '/usr/java/jdk-17.0.4.1/'
info AppiumDoctor ### Diagnostic for necessary dependencies completed, no fix needed. ###
```

Figura 7.1: Appium Doctor.

Agora estamos prontos para começar a construir os nossos testes automatizados para dispositivos móveis.

7.2 Construção de um cenário de teste

Construiremos um cenário de teste para validar a busca por livros dentro do aplicativo através da barra de pesquisas, exibida na tela principal do app.

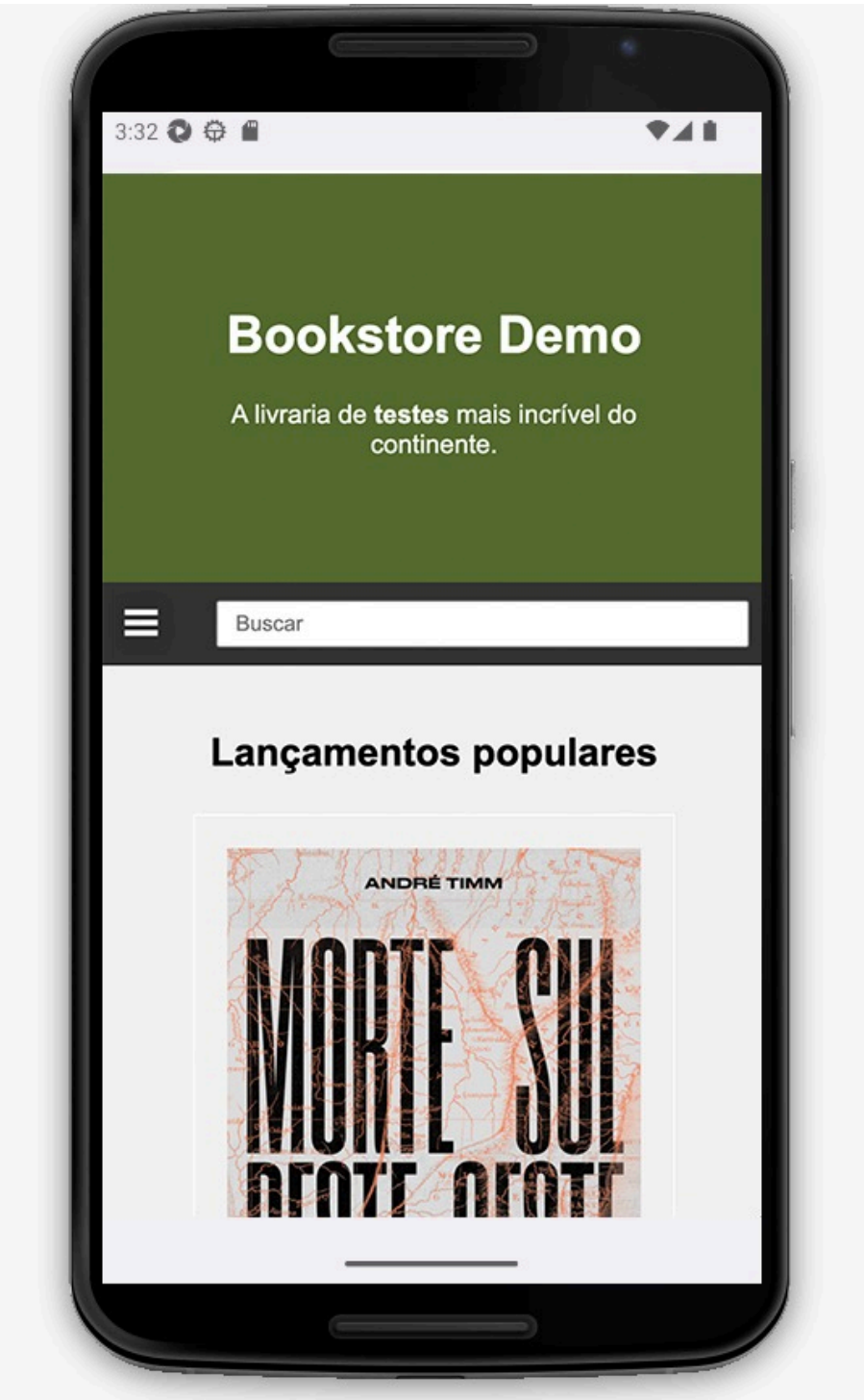


Figura 7.2: Aplicativo mobile.

O arquivo `.apk` do aplicativo deve estar dentro do projeto. Sugiro colocá-lo em `resources/apps/bookstore-demo.apk`.

Com a biblioteca do Appium instalada, o primeiro passo para construir um caso de teste mobile no Robot Framework é abrir um novo arquivo `.robot` no diretório `tests/mobile/android/`. Neste caso, daremos a ele o nome `search_test.robot`. Quando formos trabalhar no script para o aplicativo iOS, utilizaremos o diretório `tests/mobile/ios/`. Depois, podemos importar a biblioteca AppiumLibrary na seção `Settings` do arquivo.

Agora, escreveremos o passo a passo do cenário de testes, utilizando Gherkin, na seção `Test cases`.

```
[...]

Test Teardown  Close Application

*** Test Cases ***
Scenario: Search bar
  Given I open the app
   When I search for Martín Fierro
   Then I should see Martín Fierro in results screen
```

Assim como ocorreu em outros tipos de teste, devemos nos preocupar em encerrar conexões abertas ao final da execução, independentemente do resultado do teste. Para fechar o aplicativo, devemos inserir a palavra-chave `Close Application` no comando `Test Teardown`.

Na implementação da palavra-chave `I open the app` utilizaremos simplesmente a palavra-chave `Open Application` da biblioteca do Appium, que receberá uma série de parâmetros de configuração sem os quais não conseguiremos conectar o Robot ao emulador e abrir o aplicativo para interagir com ele. Minha recomendação é definir esses parâmetros no arquivo de configurações do projeto.

O principal parâmetro é o endereço do servidor do Appium, que, neste exemplo, utilizei na minha própria máquina local, na porta

`4723` ; o parâmetro `automationName` se refere ao *mobile driver* escolhido, o qual optei pelo UIAutomator; o parâmetro `app` é o caminho para o arquivo `.apk` do aplicativo sob teste, enquanto `appPackage` é o pacote deste aplicativo; `deviceName` é o nome do dispositivo móvel que está sendo emulado pelo emulador, que deve estar aberto e funcionando no momento da execução do teste automatizado; os parâmetros `platformName` e `platformVersion` se referem ao nome e versão do sistema operacional do dispositivo, neste caso, o Android na versão 13:

```
[...]

${APPIUM_SERVER}              http://localhost:4723/wd/hub
${ANDROID_AUTOMATION_NAME}    UIAutomator2
${ANDROID_APP}                ${CURDIR}/apps/app-demo.apk
${ANDROID_APP_PACKAGE}        io.bookstoredemo.app
${ANDROID_PLATFORM_NAME}      Android
${ANDROID_PLATFORM_VERSION}   %{ANDROID_PLATFORM_VERSION=13}
${DEVICE}                     Nexus_6_API_33
```

Após importar o arquivo `config.resource` em nossa suíte de testes, podemos implementar a palavra-chave que abrirá o aplicativo. Como vimos no capítulo *Escalando o projeto: arquitetura de Page Objects*, devemos abstrair as ações dos nossos scripts; então, criaremos um diretório `mobile/android/` e, dentro dele, o arquivo que conterá as ações da funcionalidade de busca de livros no aplicativo. Lá implementaremos nossas palavras-chaves de usuário para este cenário de teste.

```
[...]

*** Keywords ***
I open the app
  Open Application  ${APPIUM_SERVER}
automationName=${ANDROID_AUTOMATION_NAME}
                ...  app=${ANDROID_APP}
platformVersion=${ANDROID_PLATFORM_VERSION}
                ...  deviceName=${DEVICE}
```

```
platformName=${ANDROID_PLATFORM_NAME}
                ...
appPackage=${ANDROID_APP_PACKAGE}
```

No passo seguinte, `I search for ${query}`, executaremos duas ações: inserir o texto da busca no campo de texto e clicar no botão de busca. Para interagir com estes objetos da tela, precisamos realizar o mapeamento deles na tela. Isso é feito através do Appium Inspector, que pode ser inicializado através do Appium Desktop, clicando no botão com ícone de lupa no canto superior direito, ou diretamente através do executável, o qual pode ser baixado em https://github.com/appium/appium-inspector.

No Appium Inspector, preencheremos os dados de conexão com o Appium Server, *Remote host*, *Remote port* e *Remote path*. Depois, deveremos preencher os *desired capabilities*, que são as mesmas informações que passamos para o comando `Open Application` no código-fonte que acabamos de escrever. Esses dados podem ser definidos no formato JSON, conforme segue:

```
{
  "platformName": "Android",
  "appium:platformVersion": 13,
  "appium:deviceName": "Nexus_6_API_33",
  "appium:automationName": "UIAutomator2",
  "appium:app": "/home/{user}/Downloads/apps/app-demo.apk"
}
```

Com as configurações prontas, basta clicar no botão "Start Session" que a tela do aplicativo será exibida, conforme a imagem a seguir. Basta clicar no elemento desejado que as informações serão exibidas na lateral da tela.

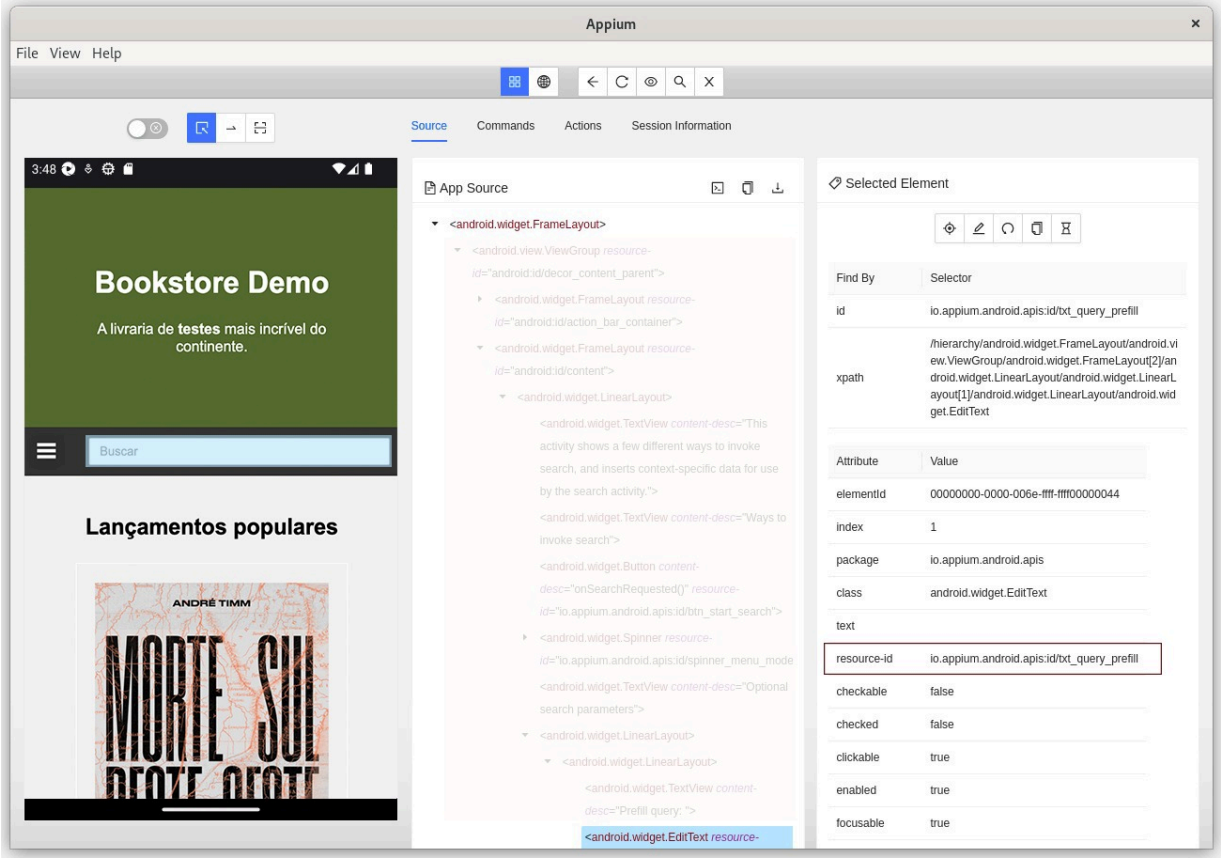


Figura 7.3: Appium Inspector.

Como destacado na imagem, é possível encontrar o identificador do objeto, ou seja, da barra de busca. Faremos o mesmo para mapear o botão de busca. Criaremos um arquivo para os *page objects* da tela inicial do aplicativo Android, no diretório `page_objects/`.

```
*** Variables ***
${SEARCH_TEXT_FIELD}  txt_query_prefill
${SEARCH_BUTTON}      btn_start_search
```

Por fim, resta-nos implementar a palavra-chave responsável pela validação de que os resultados foram devidamente exibidos na tela. Para isso, mapearemos o elemento que exibe o resultado para validar que determinado texto esperado está sendo exibido dentro deste elemento.

```
I should see ${expected} in results screen
  Wait Until Page Contains Element  ${RESULTS_LIST}
  Element Text Should Be            ${RESULTS_LIST}  ${expected}
```

Execute o teste que construímos e analise os resultados.

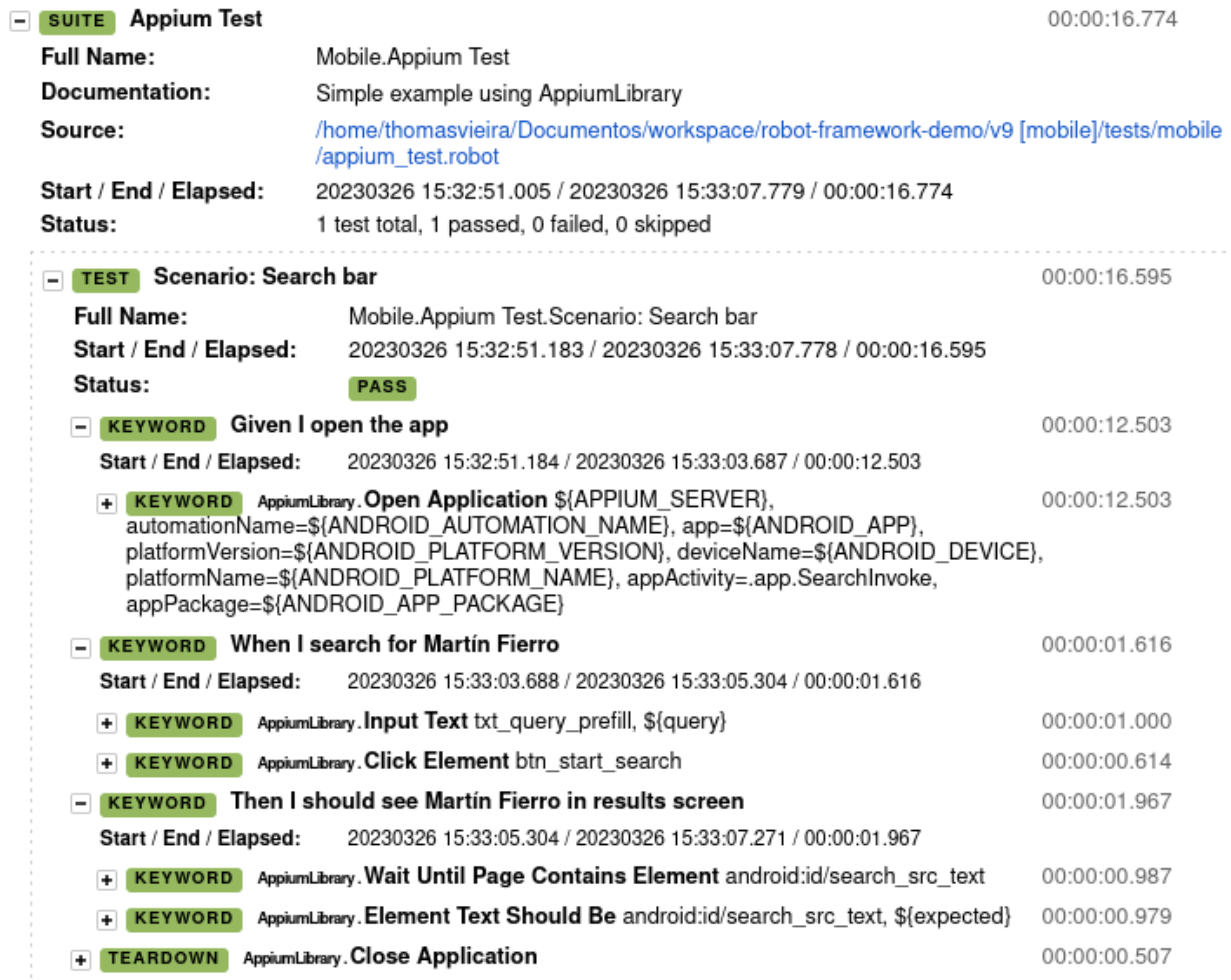


Figura 7.4: Relatório da execução.

Com isso, finalizamos este exemplo, o qual apresenta os passos introdutórios para o trabalho com automação de testes para dispositivos móveis através de Robot Framework com a biblioteca do Appium. Para trabalhar com aplicativos iOS, basta utilizar um emulador iOS e alterar as informações do *desired capabilities*. Seguindo os padrões e boas práticas, a escrita dos cenários de teste mobile não será muito diferente da escrita de testes web.

Capítulo 8
Bibliotecas do usuário

O Robot Framework é facilmente extensível. Em situações nas quais não existirem palavras-chaves ou bibliotecas que realizem determinada tarefa, é possível criar sua própria biblioteca ou estender uma biblioteca existente utilizando Python. Para trabalhar com estes recursos, é recomendável ter familiaridade com esta linguagem de programação.

Podemos supor que precisamos validar o funcionamento do histórico de livros visitados pelo cliente, aquela funcionalidade da loja virtual que apresenta um conjunto de livros com o título de "Livros que você visualizou", com intuito de explorar o interesse do cliente naquele produto.

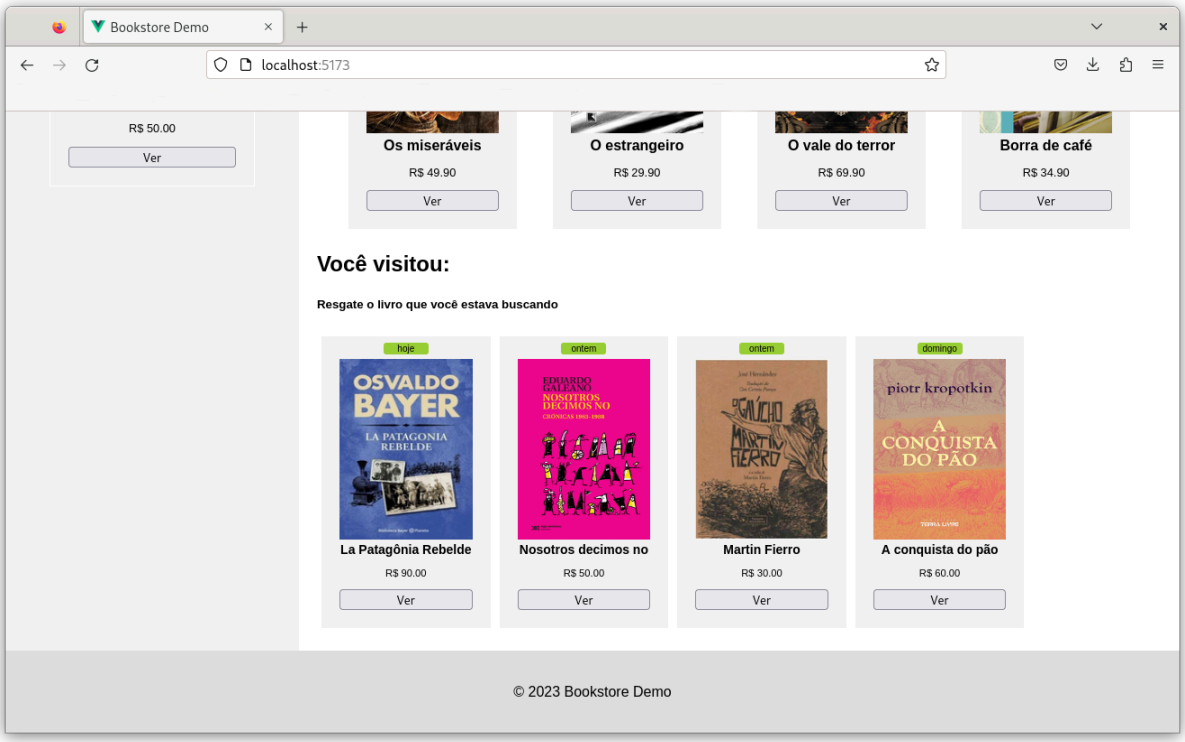


Figura 8.1: Funcionalidade de histórico de navegação.

A lógica de funcionamento deste recurso se baseia em uma estrutura de *pilha*, ou seja, o último produto visitado será o primeiro exibido no histórico. A ordem na qual os produtos aparecem é importante. Como o Robot Framework não possui essa estrutura de dados nativa, podemos implementar uma solução através da criação de uma biblioteca que emule essa estrutura de dados.

Iniciaremos criando o arquivo para a suíte de testes onde escreveremos o teste. Nele, podemos criar a seção de `Settings` como já estamos acostumados, inserindo o comando `Documentation`; a importação do arquivo de configurações globais do projeto e do arquivo de acesso a banco de dados, o qual utilizaremos para buscar massa de dados na etapa de pré-condições; e os comandos de `Test Setup` e `Test Teardown` para abrir e fechar o navegador no início e no fim do teste, respectivamente. Na seção de `Test Cases`, nomearemos o cenário, que inicia com o acesso ao site da loja virtual.

```
*** Settings ***
Documentation   Como cliente da loja virtual, eu desejo visualizar meu
histórico de livros visualizados.

Resource        ../../resources/config.resource
Resource        ../../resources/database.resource

Test Setup      Open Browser  ${URL}  ${BROWSER}
Test Teardown   Close Browser

*** Test Cases ***
Scenario: Last visited books
  Given I am on Bookstore Demo web site
```

A seguir, repetiremos três vezes o procedimento de buscar um livro diretamente da base de dados, visitar a página deste livro através da busca do site e retornar para a página inicial.

```
[...]

*** Test Cases ***
```

```
Scenario: Last visited books
  Given I am on Bookstore Demo web site
    And I get a book from database
   When I visit the book
    And I return to the home page
    And I get another book from database
    And I visit the book
    And I return to the home page
    And I get another book from database
    And I visit the book
    And I return to the home page
```

A busca do arquivo em banco de dados deve ser implementada no arquivo `database.resource`, onde uma consulta será executada buscando os dados de um livro que tenha estoque disponível. O parâmetro `returnAsDict` serve para que os dados sejam recebidos em formado de dicionário, facilitando nosso trabalho. Estes dados devem ser armazenados em uma variável de teste, nomeada com padrão ALL_CAPS, conforme recomendação do guia de estilos. Caso não exista nenhum produto cadastrado em banco de dados, utilizaremos a estrutura de tratamento de exceções para pular o teste por falta de massa de dados, registrando os motivos no *log* da execução.

Utilizei a estratégia do parâmetro embutido `${nother}` para aproveitamento da mesma palavra-chave para `I get a book from database` e `I get another book from database`, melhorando a expressividade do *script* do cenário de teste.

```
[...]

I get a${nother} book from database
  TRY
    Open database connection
    @{resultset} =     Query         SELECT id, isbn, title FROM books
WHERE stock > 0 LIMIT 1;   returnAsDict=True
    &{book_item} =     Set Variable  ${resultset}[0]
    Set Test Variable  ${BOOK}       ${book_item}
  EXCEPT
```

```
    Log   None valid product found in database.  level=WARN
    Skip  Insufficient data for testing.
  FINALLY
    Close database connection
  END
```

A busca e a visita à página do produto devem ser implementadas no arquivo `pages/home.resource`. Reaproveitaremos a palavra-chave de busca por ISBN que havíamos implementado anteriormente, aguardaremos que o produto seja completamente exibido na tela e adicionaremos esta informação à nossa pilha, que ainda não foi implementada. Deixaremos já escrita a palavra-chave `Push item to Stack`, que receberá como parâmetro o livro que desejamos inserir na pilha.

```
I visit the book
  I search for this book by ISBN
  Wait Until Page Contains          ${BOOK.title}
  Push item to Stack                ${BOOK}
```

Não se esqueça de adicionar o arquivo `pages/home.resource` aos *resources* importados na suíte, assim como o arquivo `pages/menu.page` onde implementaremos o retorno para a página inicial, que consiste simplesmente em um clique no botão "Início" da barra de menu do site.

```
I return to the home page
  Wait Until Page Contains Element  link=Início
  Click Link  Início
```

Concluiremos a escrita do cenário de teste com as validações de que a seção de histórico de produtos visitados está sendo exibida na tela principal do site, e de que os três livros visitados serão exibidos na ordem correta.

```
[...]

*** Test Cases ***
Scenario: Last visited books
```

```
  Given I am on Bookstore Demo web site
    And I get a book from database
   When I visit the book
    And I return to the home page
    And I get another book from database
    And I visit the book
    And I return to the home page
    And I get another book from database
    And I visit the book
    And I return to the home page
   Then I should see the historic session
    And I should see the last book I've visited
    And I should see the penultimate book I've visited
    And I should see the anti-penultimate book I've visited
```

Podemos implementar essas palavras-chaves em um arquivo `pages/books_browsing_history.resource`, separando os mapeamentos envolvidos em um arquivo de *page objects*. Criaremos uma palavra-chave `Pop item from Stack` para receber o item que estiver no topo da pilha e guardaremos esta informação em uma variável, a qual utilizaremos para realizar a verificação.

```
*** Settings ***
Library   SeleniumLibrary
Resource  ../page_objects/books_browsing_history_page_object.resource

*** Keywords ***
I should see the books browsing history
  Wait Until Page Contains          Resgate o livro que você estava
buscando

I should see the last book I've visited
  ${visited_book} =                 Pop item from Stack
  Element Should Contain            ${FIRST_ITEM}
${visited_book.title}

I should see the penultimate book I've visited
  ${visited_book} =                 Pop item from Stack
  Element Should Contain            ${SECOND_ITEM}
```

```
${visited_book.title}

I should see the anti-penultimate book I've visited
  ${visited_book} =                 Pop item from Stack
  Element Should Contain            ${THIRD_ITEM}
${visited_book.title}
```

No mapeamento dos objetos, utilizei *xpath* para validar a ordem de exibição dos itens:

```
*** Variables ***
${FIRST_ITEM}   xpath=//div[@id='visited-area']/div[1]
${SECOND_ITEM}  xpath=//div[@id='visited-area']/div[2]
${THIRD_ITEM}   xpath=//div[@id='visited-area']/div[3]
```

Agora desenvolveremos a biblioteca que implementará as palavras-chaves `Push item to Stack` e `Pop item from Stack` da nossa pilha. Mas como escolher qual a linguagem de programação utilizar para escrever as bibliotecas para Robot? Como o Robot Framework é escrito em Python, a linguagem mais adequada para a escrita de bibliotecas é esta, mas também é possível usar a linguagem C ou construir um *wrapper* para outras linguagens. Neste livro, trabalharemos com o Python.

Criaremos então um arquivo *.py*, que podemos chamar de `StackLibrary.py`, dentro do diretório `resources/libraries/`. A implementação pode ser feita através de um módulo ou de uma classe. Caso sejam necessários parâmetros para a importação da biblioteca, deve-se utilizar a abordagem de classes, sendo os argumentos recebidos no método construtor da classe. Caso contrário, a decisão pode ser arbitrária. Eu recomendo, por gosto pessoal, utilizar classes. Neste caso, o nome da classe será considerado o nome da biblioteca, automaticamente.

Utilizaremos alguns recursos da API do Robot, como a anotação `@library`, a qual indicará que nossa classe se trata de uma biblioteca; a anotação `@keyword`, a qual utilizaremos para indicar

quais métodos representam as palavras-chaves; e o *logger*, que servirá para registrarmos informações no *log* de execução.

É possível realizar algumas definições, como versão da biblioteca, através da definição da propriedade `ROBOT_LIBRARY_VERSION`. A mais importante definição, a qual sugiro sempre ser definida com atenção, é a do escopo da biblioteca, através da propriedade `ROBOT_LIBRARY_SCOPE`. Ela pode ser definida como:

- `TEST`, o que significa que uma nova instância será criada para cada caso de teste;
- `SUITE`, que significa que uma nova instância será criada para cada suíte de testes;
- `GLOBAL`, que significa que somente uma instância será criada durante toda execução da bateria de testes, sendo compartilhada por todos os casos de teste.

Utilizaremos o escopo `TEST` porque não queremos compartilhar o estado interno da nossa biblioteca entre diferentes casos de teste. Há ainda a propriedade `ROBOT_AUTO_KEYWORDS`, que recebe um valor booleano que indica se todos os métodos serão automaticamente acessíveis como palavras-chaves ou se somente serão os que estiverem anotados com `@keyword`. Essas definições também podem ser feitas diretamente na anotação `@library`.

```
from robot.api.deco import keyword, library
from robot.api import logger
from robot.api.exceptions import Failure


@library(scope='TEST', version='1.0', auto_keywords=False)
class StackLibrary:
```

Nossa biblioteca não precisa receber parâmetros quando inicializada, então criaremos o método construtor sem argumentos e apenas inicializaremos as duas propriedades que representarão o estado interno da nossa pilha. A primeira propriedade à qual me refiro é a lista onde armazenaremos os dados, que deve ser inicializada vazia.

A outra propriedade é um número inteiro que funcionará como um ponteiro que indicará qual é o topo da pilha, o qual será inicializado com o valor zero.

Utilizaremos o *logger* como uma boa prática, para que o usuário seja capaz de compreender o estado interno da biblioteca caso precise analisar os *logs* para *debug*. O método `logger.info()` insere mensagens informativas no relatório e surte o mesmo efeito de utilizarmos o método `print()` do Python, pois o Robot redireciona a saída padrão para o relatório. Contudo, recomendo sempre utilizar o objeto *logger*, pois torna mais claro o objetivo do código. Segundo a documentação oficial da API do Robot Framework, além de um código mais expressivo, a API registra o *timestamp* com maior acurácia.

```
[...]

@library(scope='TEST', version='1.0', auto_keywords=False)
class StackLibrary:

    pointer: int
    data: list

    def __init__(self):
        self.pointer = 0
        self.data = []
        logger.info("New stack was created.")
```

LOGGER

Essa abordagem de registrar mensagens de *log* através do `robot.api.logger` que acabamos de implementar é chamada de *Programmatic logging APIs*. Podemos enviar informações para o usuário através de mensagens no relatório de execução de forma programática, utilizando os seguintes métodos: `logger.info()`, para registrar mensagens informativas comuns; `logger.debug()`, para mensagens que só serão exibidas em caso de execução em modo *debug*; `logger.warn()`, para registrar problemas que não necessariamente causam erros ou falhas na execução do teste; `logger.error()`, para reportar um erro; e `logger.console()`, para enviar mensagens para o console. Veja a documentação desse recurso em https://robot-framework.readthedocs.io/en/latest/autodoc/robot.api.html#module-robot.api.logger.

Quando uma biblioteca possui controle de estado interno, é fortemente recomendável a criação de palavras-chaves que redefinam o estado interno. No caso de bibliotecas com escopo `SUITE` ou `GLOBAL`, esse recurso é comumente usado nos comandos de *setup* e *teardown*. Criaremos, portanto, uma palavra-chave `Clear Stack`, que limpa nossa pilha, retornando-a para seu estado inicial. Criaremos um método que não receberá parâmetros com a anotação `@keyword`, onde definiremos o nome da palavra-chave.

```
@keyword('Clear stack')
def clear(self):
    """
    Clear all items from stack, returning it to its initial state.
    """
    self.data = []
    self.pointer = 0
    logger.info("Stack is now empty.")
```

Se a palavra-chave necessitasse de argumentos, bastaria inserir na lista de parâmetros do método, tendo a possibilidade, opcionalmente, de definir explicitamente o tipo de dado do argumento. Caso seja necessário trabalhar com argumentos embutidos, adiciona-se também ao nome da palavra-chave na anotação `@keyword` os parâmetros, da mesma forma como funcionam na definição de palavras-chave na sintaxe do Robot. Assim como nos arquivos *resources*, existem diversas possibilidades ao trabalhar com argumentos. Para conhecer todas elas detalhadamente, indico a documentação oficial, disponível em https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#keyword-arguments.

Utilizaremos um parâmetro na implementação da palavra-chave `Push item to Stack`, que receberá o objeto que será empilhado. Como este objeto pode ser de qualquer tipo, não precisaríamos declarar seu tipo, mas o farei explicitamente para demonstrar como fazer.

Quando um item for adicionado à pilha, devemos inseri-lo ao fim da lista e incrementar o ponteiro em uma unidade.

```
@keyword('Push item to Stack')
def push(self, item: any):
    """
    Push an item at the top of the stack.
    """
    self.data.append(item)
    self.pointer += 1
    logger.info("Added " + str(item) + " to the top of the stack.")
```

Já a operação de remoção de um item da pilha, `Pop item from Stack`, não recebe parâmetros e deve, primeiramente, validar que a pilha não está vazia para garantir que existe algum item para retirar do topo. Então, aproveitamos o método `pop()` da classe `List` do Python, que faz exatamente o que precisamos: obter o item e armazená-lo numa variável, removendo-o da fila. Utilizamos o ponteiro para indicar qual item da lista é o que está no topo da

nossa pilha. Em seguida, devemos decrementar o valor do nosso ponteiro e retornar o item.

```
@keyword('Pop item from Stack')
def pop(self):
    """
    Return and remove the item on the top of the stack.
    """
    self.should_be_not_empty()
    item = self.data.pop(self.pointer - 1)
    self.pointer -= 1
    logger.info("Item " + str(item) + " was removed.")
    return item
```

Resta implementar o método que realiza a validação de que a pilha não está vazia. Podemos aproveitá-lo disponibilizando palavras-chaves para essa validação caso seja útil no futuro. Em geral, não recomendo escrever código que não sabemos se um dia será útil, mas, neste caso, basta adicionarmos a anotação `@keyword`. Para realizar a validação, podemos verificar se o ponteiro possui valor igual a zero ou medir a quantidade de itens presentes na lista. Qualquer uma das duas abordagens funcionará. Caso a lista esteja vazia, devemos comunicar ao Robot que algo ocorreu errado e provocar uma falha na execução.

Por padrão, exceções provocarão a falha da execução. Entretanto, é possível forçar uma falha através da exceção `Failure` ou `ContinuableFailure`, que, por sua vez, comunicará a falha sem interromper a execução do script. Também é possível provocar um erro que interrompe a execução do teste com `FatalError`, ou cancelar e pular a execução do teste com `SkipExecution`. Para as asserções que estamos implementando, comunicaremos as falhas através da exceção `Failure`.

```
@keyword('Stack should be empty')
def should_be_empty(self):
    """
    Check if stack has no item.
```

```
    """
    if len(self.data) != 0:
        raise Failure("The stack has " + str(len(self.data)) + " items.")

    logger.info("The stack is empty.")

@keyword('Stack should be not empty')
def should_be_not_empty(self):
    """
    Check if the stack has any item.
    """
    if len(self.data) == 0:
        raise Failure("The stack is empty.")

    logger.info("The stack has " + str(len(self.data)) + " items.")
```

Ao executar nosso teste, podemos verificar o funcionamento da nossa biblioteca através dos registros feitos no relatório de execução.

TEST **Scenario: Last visited books**

Full Name: Tests.Web.Visited Books Test.Scenario: Last visited books

Tags: stack

Start / End / Elapsed: 20230413 12:22:21.376 / 20230413 12:22:32.601 / 00:00:11.225

Status: PASS

+ SETUP SeleniumLibrary.**Open Browser** ${URL}, ${BROWSER}

+ KEYWORD home.**Given I am on Bookstore Demo web site**

+ KEYWORD database.**And I get a book from database**

+ KEYWORD home.**When I visit the book**

+ KEYWORD menu.**And I return to the home page**

+ KEYWORD database.**And I get another book from database**

− KEYWORD home.**And I visit the book**

Start / End / Elapsed: 20230413 12:22:29.643 / 20230413 12:22:29.994 / 00:00:00.351

+ KEYWORD home.**I search for this book by ISBN**

+ KEYWORD SeleniumLibrary.**Wait Until Page Contains** ${BOOK.title}

− KEYWORD StackLibrary.**Push item to Stack** ${BOOK}

Documentation: Push an item at the top of the stack.

Start / End / Elapsed: 20230413 12:22:29.993 / 20230413 12:22:29.994 / 00:00:00.001

```
12:22:29.993  INFO  Added {'id': '6370', 'isbn': '978-0-7930-3183-2', 'title': 'Him
                    imagine although growth.'} to the top of the stack.
```

+ KEYWORD menu.**And I return to the home page**

+ KEYWORD database.**And I get another book from database**

+ KEYWORD home.**And I visit the book**

+ KEYWORD menu.**And I return to the home page**

+ KEYWORD books_browsing_history.**Then I should see the books browsing history**

− KEYWORD books_browsing_history.**And I should see the last book I've visited**

Start / End / Elapsed: 20230413 12:22:31.498 / 20230413 12:22:31.531 / 00:00:00.033

− KEYWORD ${visited_book} = StackLibrary.**Pop item from Stack**

Documentation: Return and remove the item on the top of the stack.

Start / End / Elapsed: 20230413 12:22:31.499 / 20230413 12:22:31.499 / 00:00:00.000

```
12:22:31.499  INFO  The stack has 3 items.
12:22:31.499  INFO  Item {'id': '6370', 'isbn': '978-0-7930-3183-2', 'title': 'Him
                    imagine although growth.'} was removed.
12:22:31.499  INFO  ${visited_book} = {'id': '6370', 'isbn': '978-0-7930-3183-2',
                    'title': 'Him imagine although growth.'}
```

+ KEYWORD SeleniumLibrary.**Element Should Contain** ${FIRST_ITEM}, ${visited_book.title}

+ KEYWORD books_browsing_history.**And I should see the penultimate book I've visited**

+ KEYWORD books_browsing_history.**And I should see the anti-penultimate book I've visited**

+ TEARDOWN SeleniumLibrary.**Close Browser**

Figura 8.2: Logs do uso da Stack Library.

Assim, concluímos a implementação da nossa Stack Library, que simula a pilha que viabiliza o teste do histórico de navegação nos livros da loja virtual. Para conhecer todos os recursos e distintas possibilidades de estender o Robot Framework e criar novas bibliotecas, leia mais em https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#extending-robot-framework.

8.1 Libdoc

Perceba que criei uma pequena documentação das palavras-chaves da biblioteca utilizando *doc strings*, mesmo que bastante simples. Isso foi feito para que possamos gerar uma documentação para nossa biblioteca dentro dos padrões do Robot Framework, o que será muito útil caso desejemos disponibilizá-la para uso de terceiros. A ferramenta embutida no Robot para esta tarefa é a Libdoc.

A Libdoc também pode ser utilizada para documentar o próprio projeto de testes, com suas suítes e recursos. Contudo, é mais comum sua utilização para documentação de bibliotecas.

Para gerar a documentação, basta utilizarmos o comando `libdoc [options] library_or_resource output_file`, onde `library_or_resource` é o arquivo ou nome da biblioteca para o qual queremos gerar a documentação e `output_file` é o arquivo onde desejamos salvar o arquivo de saída. Utilizaremos duas definições de opções: definiremos o formato de saída através da *flag* `-f` ou `--format`, a qual pode ser HTML, XML ou JSON; e o tema escuro com a *flag* `--theme`, a qual pode assumir valor `dark` ou `light`.

```
libdoc -f html --theme dark  resources/libraries/StackLibrary.py
docs/StackLibraryDocs.html
```

Abrindo o arquivo gerado, podemos visualizar a documentação gerada em um formato já familiar para quem está acostumado com o Robot Framework:

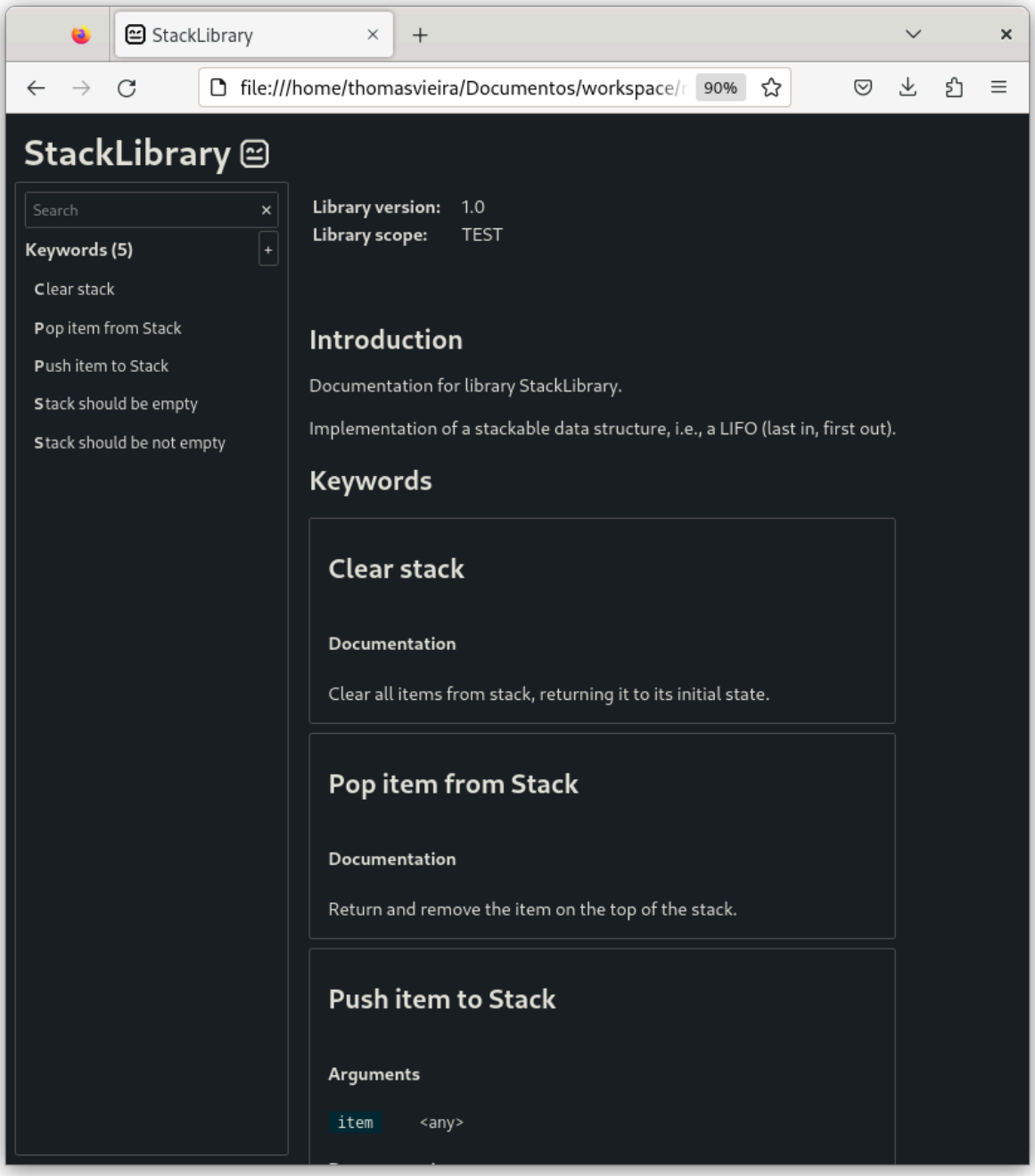


Figura 8.3: Documentação da StackLibrary.

Na sintaxe padrão, é possível utilizar itálicos com `_lorem ipsum_` e negrito com `*lorem ipsum*`. URLs são automaticamente convertidas em *links*. Entretanto, também é possível alterar o formato da documentação para HTML, através da opção `--docformat HTML`, e escrever o conteúdo do *doc string* nesta sintaxe, o que pode facilitar quando for necessário criar tabelas e listas.

Para saber mais sobre a sintaxe e as possibilidade da Libdoc, acesse https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#libdoc

8.2 Listener interface

O Robot possui uma interface de *listeners* que pode ser utilizada para, por exemplo, enviar dados para obtenção de métricas sobre os testes, envio de notificações por e-mail ou aplicativos de mensagem, ou integrações com outros sistemas. Esse recurso permite que uma biblioteca execute ações no início ou no final de uma suíte de testes; no início ou no final de um caso de teste; quando algum recurso for importado; quando os *logs* forem gerados, dentre outras possibilidades.

Criei um arquivo `LogListener.py` para demonstrar como utilizar os principais métodos da interface *listener*:

```
ROBOT_LISTENER_API_VERSION = 3

def start_suite(data, result):
    print("Iniciada suíte de testes: ", data.name)

def start_test(data, result):
    print("Iniciado caso de teste: ", data.name)

def end_test(data, result):
    if not result.passed:
```

```
        print('Test "%s" failed: %s' % (result.name, result.message))
        # Aqui poderíamos enviar dados sobre a falha para algum sistema
externo.

def end_suite(data, result):
    print("Fim da suíte.")
```

Para executar, basta utilizar o parâmetro `--listener` na linha de comando:

```
$ robot -d reports/ --listener resources/libraries/LogListener.py  tests/
```

Todos os detalhes dessa funcionalidade estão em https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#listener-interface.

CAPÍTULO 9
Boas práticas e guia de estilos

Escrever código limpo, seguir boas práticas e implementar padrões de desenvolvimento são atitudes sensatas e recomendadas para todo e qualquer profissional da área de engenharia de software. Não obstante, essas preocupações são ainda mais essenciais no caso de quem desenvolve testes automatizados, devido ao fato de que, se o sistema de produção sofre atualizações, então provavelmente o código do teste necessitará de manutenção, especialmente quando se trata de teste de interface. Isso significa que é comum que o código-fonte dos testes automatizados sofra manutenção frequentemente. Se a manutenção do teste for complexa demais, logo a equipe não terá tempo para mantê-los e deixará de executá-los.

Segundo Robert C. Martin, no capítulo sobre testes em seu livro *Código limpo*, existem três aspectos que tornam um teste limpo: legibilidade, legibilidade e legibilidade. É isso que deve nortear todo programador ou programadora de testes automatizados. Com os conceitos apresentados aqui, buscamos simplicidade, clareza e coerência. Um projeto com essas características tem boas chances de possuir boa legibilidade e manutenibilidade.

Neste capítulo, falaremos sobre algumas práticas recomendadas para o desenvolvimento de projetos de automação em geral e também para projetos específicos com Robot Framework.

9.1 Guia de estilos

Sobre convenções e padrões de codificação, seguiremos as diretrizes do guia de estilos do Python, tecnologia na qual o Robot é

desenvolvido. O guia de estilos está no PEP 8, ou *Python Enhancement Proposals 8*, no endereço https://peps.python.org/pep-0008/. Aqui citaremos apenas as diretrizes essenciais.

Nomenclaturas

Sempre utilize nomes descritivos, mas concisos. Nomes de casos de teste devem expressar exatamente a regra de negócio que está sendo validada e as palavras-chaves devem expressar exatamente o que ela faz. Em ambos os casos, se o nome ficar muito longo ou possuir a conjunção "e", temos um indício de que o código deve ser refatorado e dividido. Segundo a recomendação do livro *Código Limpo*, devemos implementar testes que validem um único conceito, evitando scripts longos que testem várias coisas diferentes em sequência.

Os nomes das suítes de testes merecem especial atenção, pois são criados automaticamente pelo Robot com base nos nomes dados aos diretórios e arquivos de teste. As extensões dos arquivos são ignoradas, *underlines* são convertidas para espaços e as palavras são capitalizadas. Por exemplo, `music_player_test.robot` receberá a nomenclatura de "Music Player Test". Não esqueça do sufixo "_test".

Variáveis de escopo global e constantes devem seguir o padrão ALL*CAPS, por exemplo,* `${TIMEOUT}` *e* `${ACCESS_KEY}` *. Variáveis comuns e argumentos de palavras-chaves devem seguir o padrão* _*snake_case*, por exemplo, `${text}` e `${phone_number}` . Arquivos e diretórios também devem seguir o padrão *snake_case*.

Espaços ou tabs?

Recomendo utilizar espaços em vez de *tabs*, que pode causar confusão dependendo do editor de código e sistema operacional. Defina se utilizará dois ou quatro espaços para separar e alinhar as colunas do código-fonte e respeite essa decisão do início ao fim do projeto. Alinhar comandos e parâmetros em colunas pode não

parecer bonito para quem está acostumado a desenvolver em linguagens de programação comuns, mas em projetos de Robot Framework essa atitude facilita (e muito) a leitura.

Evite espaços e linhas em branco desnecessários. Na realidade, evite tudo que possa parecer não essencial. Por exemplo, antes de utilizar estruturas complexas de controle de fluxo, busque por palavras-chave que façam o mesmo trabalho diretamente.

Imports

A importação de bibliotecas e recursos externos deve ser organizada nesta ordem: bibliotecas padrão; bibliotecas de terceiros; bibliotecas do usuário. Recomenda-se separar esses grupos com uma linha em branco.

O local correto de importar uma biblioteca no projeto é onde seus recursos serão utilizados. Não coloque uma lista extensa de bibliotecas no arquivo da suíte de teste, por exemplo, se as palavras-chave dessas bibliotecas forem utilizadas somente nos arquivos de *resource*. A biblioteca Faker, por exemplo, provavelmente não precisará ser importada em nenhum outro lugar além de no próprio componente de Dataprovider. Ele encapsula todos os seus recursos.

Comentários

Evite comentários. O Robot provê outras formas melhores de inserir informações sobre o teste, como os comandos `Documentation` e `Metadata`. Além do mais, um bom código-fonte dispensa comentários.

Código-fonte comentado jamais deve ser versionado no repositório do projeto. Apague-o, pois se for necessário recuperá-lo, ele estará no histórico do próprio repositório.

9.2 Esperas

Existem distintas maneiras de lidar com *timeouts* e esperas, especialmente em relação ao Selenium. Vejamos nesta seção quais são as recomendações.

É importante lembrar que o Selenium considerará os segundos como a unidade de tempo padrão caso outra unidade não seja especificada. O ideal é explicitar a unidade de tempo, por exemplo, `10 seconds`, `1.5 min`, `1 day 2 hours` ou `5 milliseconds`. Mais detalhes sobre como o Robot Framework lida com unidades de tempo, você pode conferir em http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#time-format.

Timeout

É possível redefinir o *timeout* padrão utilizado pelo Selenium, através do comando `Set Selenium Timeout`, e também o tempo máximo de espera pelo carregamento de uma página, através da palavra-chave `Set Selenium Page Load Timeout`. Caso a desenvolvedora não defina essas valores, o padrão será de cinco segundos. Muitas palavras-chaves possuem o parâmetro opcional `timeout` e utilizarão esses parâmetros caso nenhum valor seja definido como argumento.

Espera incondicional versus esperas condicionais

Evite utilizar a espera incondicional `Sleep`, exceto em casos de *debug*. Utilize as esperas condicionais do Selenium, que são as palavras-chaves que normalmente começam por "Wait until...". Esperas incondicionais tornam a execução mais lenta do que o necessário. Este tempo pode não parecer significativo ao executar apenas um caso de teste, mas na execução de conjuntos maiores certamente o tempo total será maior do que o aceitável.

As palavras-chaves para esperas disponibilizadas pelo Selenium incluem: espera até que um elemento contenha ou não contenha

determinado texto; espera até que um elemento esteja ou não visível; espera até que a página contenha ou não determinado elemento ou determinado texto, dentre outras. Consulte a documentação das palavras-chaves do Selenium para obter a lista completa. Caso nenhuma delas solucione o problema, ainda há a palavra-chave `Wait for condition`, a qual permite que seja utilizada uma condição escrita em JavaScript para acessar elementos da página ou do navegador. Por exemplo, para aguardar até que o título da página seja "Foo", poderíamos utilizar como condicional a expressão `return document.title == "Foo"`.

Chamamos esses comandos de "esperas condicionais" e também de "esperas explícitas", pois são esperas explicitamente definidas para ocasiões específicas. Quando estivermos lidando com sistemas muito lentos e precisarmos definir uma espera maior pelo carregamento de todos os elementos, podemos utilizar a "espera implícita", que consiste em definir um tempo de espera para que o robô busque um elemento na página antes de provocar falha por não o encontrar. Essa definição é feita através da palavra-chave `Set Selenium Implicit Wait`.

9.3 Tags

Para facilitar a execução de determinados conjuntos de casos de teste, ou até mesmo de casos de teste específicos, podemos agrupá-los utilizando *tags*. Enquanto estamos desenvolvendo um novo caso de teste, é normal executarmos somente o script em questão inúmeras vezes e, para não perdermos tempo executando a suíte inteira, podemos utilizar uma *tag*. O Robot disponibiliza o argumento `--test` para especificarmos os casos de teste que desejamos executar, mas utilizar *tags* é mais prático.

Sendo assim, podemos adicionar a *tag* `natural_person` ao cenário de teste de cadastro de pessoa física e a tag `juridical_person`, utilizando

o comando `[Tags]` no início de cada cenário de teste, além de adicionar a tag `register` à nossa suíte, utilizando o comando `Default tags` na sessão de configurações do arquivo.

```
*** Settings ***
Documentation   Como cliente da loja virtual, desejo criar um
          ...   cadastro no sistema para poder visualizar meus
          ...   pedidos e lista de desejos.
Test Tags    register   embedded_args

[...]

*** Test Cases ***
Scenario: Success sign in as Natural Person
    [Tags]  natural_person

    [..]

Scenario: Success sign in as Juridical Person
    [Tags]  juridical_person

    [...]
```

Além de facilitar o comando de execução, classificar os cenários de teste através do uso de *tags* pode ser muito útil para visualizar estatísticas e informações geradas automaticamente nos relatórios de execução com base nos grupos classificados com *tags*.

Para determinar quais *tags* devem ser incluídas na execução, utilizamos a configuração `--include` (ou `-i` , em sua forma reduzida), seguida da *tag* que desejamos. Caso seja necessário determinar *tags* que não queremos executar, utilizamos a configuração `--exclude` (ou `-e` , em sua forma reduzida), seguida das *tags* que não desejamos executar.

Com o comando a seguir, somente o cenário de teste de cadastro de pessoa jurídica será executado:

```
$ robot -d reports/ --include juridical_person tests/
```

Podemos combinar diferentes *tags* na mesma execução utilizando os operadores `AND`, `OR` e `NOT` — por exemplo, para executar somente os cenários de teste com a *tag* `register` e que não possuam a *tag* `natural_person`:

```
$ robot -d reports/ --include registerNOTnatural_person tests/
```

É possível forçar o Robot a ignorar determinado cenário de teste utilizando a *tag* `robot:skip`. Assim, o cenário aparecerá no relatório de execução, mas o teste não será executado. Existem outras *tags* reservadas como esta. A lista completa está em https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#reserved-tags.

Esses são os principais recursos para execução dos scripts, os quais utilizaremos na imensa maioria das ocasiões. Você pode ver todas as possibilidades de parametrização da execução em https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#configuring-execution.

Capítulo 10
Para saber mais

Este livro trouxe uma visão prática bastante completa, mas vale saber que ainda há outras possibilidades para o Robot Framework. Sempre é possível se aprofundar ainda mais, e a disponibilidade de conteúdo é abrangente. Aqui estão algumas dicas de onde continuar buscando conhecimento.

Praticar é a melhor forma de fixar os aprendizados. Lembre-se de que o código-fonte de todos os exemplos utilizados neste livro está no repositório https://github.com/thvieira/robot-framework-demo. Utilize-os à vontade.

Conteúdo

O melhor lugar para encontrar conteúdo de qualidade e confiável sobre Robot Framework é sua própria documentação oficial, na qual me baseei para escrever este livro. Ela está disponível em https://docs.robotframework.org/docs.

Existe um *playground* on-line no site oficial do Robot Framework para familiarização com a ferramenta. Acesse: https://www.robotframework.org/code.

Comunidade

O Robot Framework possui uma comunidade bastante ativa. Existe uma conferência internacional específica dessa comunidade, chamada Robocon, onde se apresentam as novidades das últimas versões e se discutem as próximas funcionalidade, bem como formas de difundir o aprendizado na ferramenta. O site da conferência é https://robocon.io/.

Participe de discussões e compartilhe soluções colaborativamente com a comunidade no fórum oficial em https://forum.robotframework.org.

Outros tipos de teste

Existem muitos outros tipos de testes com Robot Framework, como SAP, Desktop, dentre outros. As principais bibliotecas existentes estão listadas na página https://robotframework.org/#resources. Caso a ferramenta que você procura não esteja presente na lista, uma busca no Google pode levar você até outras bibliotecas diversas que são disponibilizadas pela própria comunidade. A organização e escrita dos testes será muito parecida com o que aprendemos neste livro.

Finalmentes

Meu desejo é que este livro tenha lhe ajudado a ser um(a) profissional tecnicamente melhor. Caso tenha *feedbacks* para compartilhar, não hesite em me escrever. Obrigado por chegar até aqui!

Capítulo 11
Referências

ANICHE, Maurício. *Testes automatizados de software*: Um guia prático. São Paulo: Casa do Código, 2015.

BULATY, Wojciech. Testing Microservices: an Overview of 12 Useful Techniques - Part 1. *InfoQ*, 2019. Disponível em: https://www.infoq.com/articles/twelve-testing-techniques-microservices-intro/. Acesso em 6 jul. 2023.

EMERY, Dale H. Writing Maintainable Automated Acceptance Tests. *Conversations with Dale*, 2009. Disponível em https://cwd.dhemery.com/2009/11/wmaat/. Acesso em: 26 jun. 2023.

FOWLER, Martin. *Refatoração*: Aperfeiçoando o design de código existente. Porto Alegre: Bookman, 2020.

FOWLER, Martin. Contract Test. *MartinFowler.com*, 2011. Disponível em www.martinfowler.com/bliki/ContractTest.html.

Acesso em: 26 jun. 2023.

FOWLER, Martin. Test Coverage. *MartinFowler.com*, 2012a. Disponível em www.martinfowler.com/bliki/TestCoverage.html. Acesso em: 26 jun. 2023.

FOWLER, Martin. Test Pyramid. *MartinFowler.com*, 2012b. Disponível em www.martinfowler.com/bliki/TestPyramid.html. Acesso em: 26 jun. 2023.

FOWLER, Martin. Page Object. *MartinFowler.com*, 2013. Disponível em www.martinfowler.com/bliki/PageObject.html. Acesso em: 26 jun. 2023.

GÄRTNER, Markus. *ATDD by Example*: A Practical Guide to Acceptance Test-Driven Development. Upper Saddle River, NJ: Addison-Wesley Professional, 2012.

MARTIN, Robert C. *Clean Code*: A Handbook of Agile Software Craftsmanship. 1. ed. Upper Saddle River, NJ: Prentice Hall, 2008.

MARTIN, Robert C. *The Clean Coder*: a Cod of Conduct for professional. 1. ed. Upper Saddle River, NJ: Prentice Hall, 2011.

VOCKE, Ham. The Practical Test Pyramid. *MartinFowler.com*, 2018. Disponível em https://martinfowler.com/articles/practical-test-pyramid.html. Acesso em: 26 jun. 2023.

Links

Arquivo de conteúdos da Robocon: https://robocon.io/archive.

Documentação oficial do Robot Framework e suas bibliotecas: https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

Documentação oficial do Selenium: https://www.selenium.dev/documentation/.

Referencial da sintaxe Gherkin no site do Cucumber: https://cucumber.io/docs/gherkin/reference/.

Hub de conceitos de computação em nuvem: https://aws.amazon.com/pt/what-is.

Python Enhancement Proposals 8 (PEP 8): https://peps.python.org/pep-0008/.

W3Schools: http://www.w3schools.com.

Capítulo 12
Anexo I: Técnicas de mapeamento de elementos HTML

O script de teste automatizado interage com os elementos do sistema através de ações como clique, duplo clique, inserção de texto, dentre outras. Para identificar o elemento com o qual o script vai interagir, é necessário que o testador mapeie este elemento na página, utilizando alguma das técnicas disponíveis no Selenium para construir um localizador. Esta é uma das etapas mais importantes na construção de testes automatizados.

As técnicas de mapeamento dos elementos mais utilizadas são os atributos ID e NAME ou uma expressão de XPATH, portanto, são as estratégias padrão dos comandos do Selenium. Cada estratégia de mapeamento possui vantagens e desvantagens. Utilizar o atributo ID é o recomendado sempre que possível, pois sua sintaxe é simples e o processamento de busca do elemento nos navegadores será mais veloz. Normalmente, os parâmetros de comandos que se referem a localizadores de elementos são nomeados como *"locator"*. Os três conjuntos de estratégias são *Default locator strategy*, *Explicit locator strategy* e *Implicit locator strategy*. Analisaremos detalhadamente todas as possibilidades.

Default locator strategy

Quando omitirmos a especificação de qual estratégia aplicar, o robô utilizará por padrão os atributos ID ou NAME. Chamamos isso de *Default locator strategy*. Alguns comandos aceitarão outras estratégias por padrão, dependendo de seu objetivo. O comando `Click link`, por exemplo, considerará adicionalmente o atributo HREF e o texto do link sem necessidade de indicação explícita no código.

```
Click Element   example    # Considera o valor dos atributos ID ou NAME.
Click Link      example    # Considera também o texto do link ou o
atributo HREF.
Click Button    example    # Considera o valor dos atributos ID, NAME ou
VALUE.
```

Implicit locator strategy

Caso o localizador comece com os caracteres `//`, então a estratégia de busca por XPATH será implicitamente assumida pelo robô. Utilizar `//h1` é equivalente à forma explícita `xpath: //h1`.

```
Click Element   //button[text()="Open"]
Submit Form     //div[@id="main"]/form
```

Montar expressões com XPATH pode não parecer trivial inicialmente, mas nos salvará muitas vezes — incrivelmente, ainda existem times de desenvolvimento que são resistentes a dedicar tempo em adicionar atributos ID ou NAME aos elementos HTML, alegando falta de tempo e dificultando a vida do testador.

Optamos pela abordagem do XPATH quando nos depararmos com mapeamentos mais complexos. Recomendo fortemente o tutorial de XPATH do W3schools, disponível em www.w3schools.com/xml/xpath_intro.asp.

Explicit locator strategy

Neste caso, a estratégia é definida explicitamente pelo testador no momento de informar o localizador. A definição é feita através do nome da estratégia como prefixo do localizador, com a sintaxe `strategy: value` ou `strategy = value`. As duas opções são equivalentes, mas a primeira opção é recomendada para evitar confusão, pois a sintaxe com o separador `=` é idêntica à sintaxe de argumentos nomeados do Robot Framework.

Estratégia	Funcionamento	Exemplo
id	Atributo `id`.	id: exemplo
name	Atributo `name`.	name: exemplo
identifier	Atributos `id` ou `name`.	identifier: exemplo
class	Atributo `class`.	class: exemplo
tag	Nome da *tag*.	tag: h1
xpath	Expressão XPATH.	xpath: //div[text()="exemplo"]
css	Seletor CSS.	css: p#exemplo
dom	Expressão DOM.	dom: document.images[5]
link	Texto exato da tag `<a>`.	link: Clique aqui
partial link	Parte do texto da tag `<a>`.	partial link: aqui
data	Atributos `data-...`.	data: message-id=00EX1
jquery	Expressão jQuery.	jquery: div.exemplo
default	Estratégia padrão.	default: exemplo

Algumas dessas estratégias são pouco conhecidas, como a expressão DOM, por exemplo, que se trata do objeto `document` da página e que dispõe muitas informações sobre o documento HTML da página e sobre a janela do navegador. O W3Schools também possui um tutorial muito bom para quem deseja aprender a trabalhar com DOM, disponível em https://www.w3schools.com/jsref/dom_obj_document.asp.

Adicionalmente, é possível criar estratégias de mapeamento personalizadas, através do comando `Add Location Strategy`. Veja como funciona a implementação em https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html#Custom%20locators.

WebElements

Também é possível utilizarmos o objeto `WebElement` do Selenium como parâmetro no lugar dos localizadores. Essa abordagem requer que o elemento seja mapeado previamente através do comando `Get WebElement`, que recebe como parâmetro o localizador e retorna o objeto mapeado para uma variável que poderá ser utilizada posteriormente em substituição aos localizadores como parâmetro dos comandos.

```
${elemento} =    Get WebElement    id: exemplo
Click Element    ${elemento}
```

Mapear os objetos de uma página web requer conhecimentos básicos de HTML, a linguagem de marcação padrão da web. Caso queira aprender mais sobre este assunto, recomendo o tutorial do W3Schools, disponível em http://www.w3schools.com/html/. O conteúdo deste tutorial é baseado na documentação oficial do Selenium para Robot Framework, disponível em http://robotframework.org/SeleniumLibrary/SeleniumLibrary.html#Locating%20elements.

CAPÍTULO 13
Anexo II: Camadas de abstração para projetos de automação

Neste anexo, apresentarei como organizo as camadas de abstração dos meus projetos de automação, sem atrelar a uma tecnologia específica e sem me propor a detalhar os conceitos do padrão de arquitetura *page object*, assumindo que você já tenha familiaridade com ele após a leitura do livro até aqui. As ideias que serão expostas agora não se aplicam somente para interface web, mas para qualquer outra tecnologia de interface que possa ser automatizada, bem como mobile, Java Swing, Desktop, dentre outras.

Scripts automatizados para um sistema que será submetido a atualizações sempre necessitarão de manutenção. É nossa responsabilidade lidar com isso e pensar no design de nosso projeto, preparando o código-fonte para que essas tarefas, que normalmente são executadas por outros(as) profissionais, se tornem o mais simples possível. Códigos confusos e de difícil alteração tornam-se um estorvo para a equipe e provavelmente serão abandonados rapidamente após as primeiras atualizações do sistema, devido ao acúmulo de alterações necessárias.

Separo as responsabilidades dos componentes em meus projetos, independente da linguagem, ao menos em três níveis:

1. Os **elementos** da página ou aplicação, que é a camada de *page objects*;
2. As **ações** das funcionalidades do sistema, que é a camada por vezes chamada de *tasks*, outras vezes chamada de *resources*, ou algum termo equivalente;
3. Os **cenários de teste** escritos em linguagem humana amigável, que é a camada de *test cases*.

Essa é a estrutura básica que todo e qualquer projeto de automação de interface deve seguir. Isso facilita que outras pessoas programadoras consigam dar manutenção e continuidade ao projeto por estarem familiarizadas com sua arquitetura, mesmo que não estejam acostumadas com determinada linguagem de programação. Além disso, o código fica mais legível e reaproveitável, aspectos que também melhoram a manutenibilidade dos scripts.

Cada camada possui uma responsabilidade e uma razão de existir. Falaremos um pouco sobre cada uma delas.

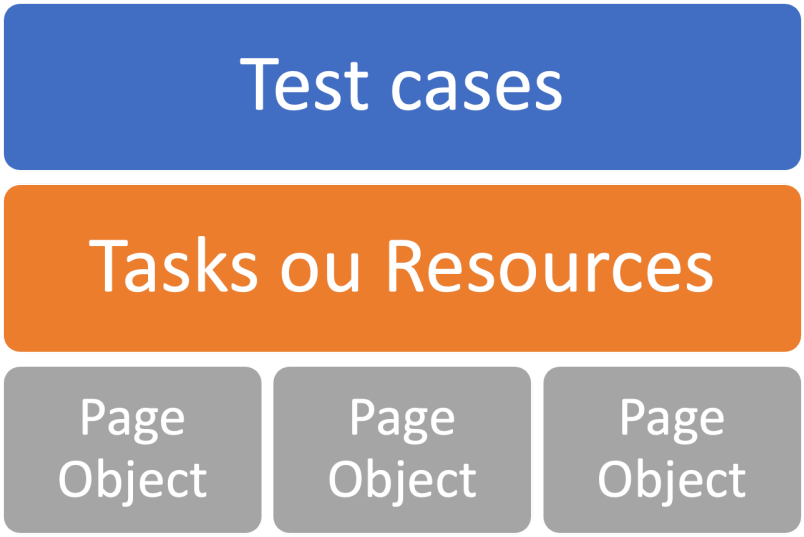


Figura 13.1: Arquitetura de Page Objects.

A responsabilidade da camada de *page objects* é prover os **elementos** de uma página, encapsulando seus mapeamentos. Esses elementos, também chamados de *objetos*, podem ser botões, campos de texto, imagens, *checkboxes*, dentre outros. Para melhor organização e manutenibilidade, criamos um arquivo para cada página ou tela do sistema ou aplicação, por exemplo, `LoginPageObject`, `ShoppingCartPageObject`, `ContactPageObject` ou `MyWishListPageObject`. Não deve haver dados, ações ou validações dentro dos *page objects*.

Avançando um nível de abstração, chegamos à camada intermediária, a das *tasks* — ou *resources*, ou outras nomenclaturas que eventualmente podem variar. A responsabilidade aqui é encapsular as **ações** do sistema, divididas por funcionalidades, como `CheckoutTasks` ou `LoginTasks`. São esses métodos que fazem a interação com os elementos dos *page objects*. É nesta parte que existirão maiores nuances de modelagem por parte da pessoa programadora, que optará por criar ações mais ou menos genéricas, como "realizar login" ou "preencher nome de usuário", "preencher senha" e "clicar no botão *Entrar*". Observo que, quanto mais atômica a ação, maior a possibilidade de reaproveitamento do código. Não deve haver dados dentro das *tasks/resources*. Tenha em mente que, alterando os dados, um fluxo de "realizar login" pode ser reutilizado para cenários de sucesso ou falha por *username* incorreto, senha incorreta, usuário inativo ou dados inválidos, por exemplo.

Chegamos à camada de abstração mais alta até então, a camada dos *test cases*, os quais, não raro, estarão contidos dentro de uma *test suite*. Aqui o código-fonte deve ser facilmente legível, e, se possível, escrito em linguagem amigável para o ser humano, como na sintaxe Gherkin, por exemplo. Desta forma, ao ler o caso de teste, uma pessoa deve conseguir, sem dificuldades, compreender o passo a passo de ações realizadas.

Todavia, ainda podemos evoluir este modelo simplificado criando algumas outras camadas de responsabilidades. Citarei mais duas: uma para isolar os pontos de verificação dos scripts, chamada justamente *verification points*, e outra para os dados de teste, que, dependendo do contexto do projeto, poderia ser um gerador de dados (*dataprovider*), ou um conjunto de dados pré-armazenados (*datapool*), ou um acesso de leitura em banco de dados (*database*), ou requisições para uma API de serviços (*services*) etc.

Os pontos de verificação funcionam como as *tasks*, encapsulando ações e interagindo com elementos da tela; contudo, a sua responsabilidade é representar ações específicas de **validação**. É comum encontrarmos os pontos de verificação implementados junto

das *tasks*, mas convém separá-los em projetos maiores. Sem validação, o script automatizado não deveria ser considerado um teste afinal de contas, testes servem para validar alguma coisa. Por isso, a etapa de validação é indispensável.

Geralmente, os **dados de teste** são o principal problema para quem desenvolve testes automatizados. A técnica de como acessar dados que contemplem pré-requisitos varia muito em cada time e empresa. Esses dados podem ser obtidos através de consultas de leitura em banco de dados, requisição a serviços, *mocks* etc. No caso de dados livres de contexto, a tarefa é facilitada, de maneira que podemos gerar dados dinamicamente através de bibliotecas como o Faker, que são capazes de gerar nomes, números de documentos, endereços de correio eletrônico e muito mais. Existem implementações de bibliotecas Faker em diversas linguagens, como JavaScript, Python etc., inclusive para Robot Framework.

O fato é que misturar acesso ou geração de dados com as *tasks* ou *test cases* provocará confusão e redundância de código. Então, em vez de criar *tasks* para "realizar login com sucesso", "realizar login com nome de usuário inválido" e "realizar login com senha incorreta", deve-se criar apenas uma *task* "realizar login" que receberá os dados através da invocação nos três casos de teste, que apenas repassam os dados para a *task*. A responsabilidade de gerar ou buscar esses dados deve estar encapsulada em uma camada de *dataprovider*, *datapool*, *database* ou *service*, dependendo do caso.

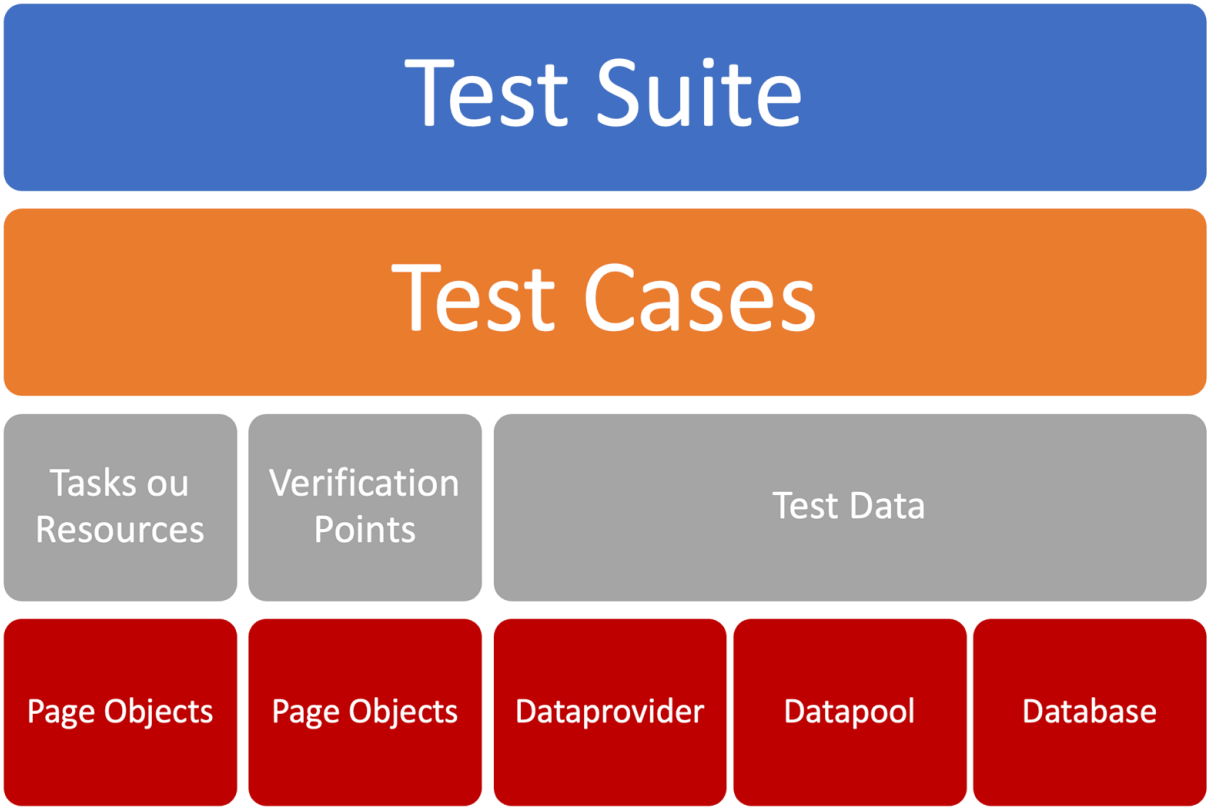


Figura 13.2: Arquitetura mais completa.